Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Mladovan

Paralelizacija Gram-Schmidtovega postopka na sistemu Intel Xeon Phi

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2016

Rezultati magistrskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov magistrskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Domen Mladovan, vpisna številka 63060204, avtor zaključnega dela z naslovom:

Paralelizacija Gram-Schmidtovega postopka na sistemu Intel Xeon Phi (angl. *Parallelization of Gram-Schmidt algorithm on Intel Xeon Phi system*)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 30. november 2016 Podpis študenta/-ke:

Zahvaljujem se doc. dr. Boštjanu Slivniku za pomoč in usmerjanje pri izdelavi magistrskega dela. Zahvala gre tudi vsem bližnjim za podporo med študijem in pisanjem magistrske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPU	Cental Processing Unit	centralna procesna enota
GPU	Graphics Processing Unit	grafična procesna enota
MIMD	Multiple Instruction Mulitple Data	paralelna arhitektura računalniškega sistema po Flynnovi taksonomiji, kjer več neodvisnih procesnih enot izvaja operacije nad različnimi toki podatkov
CGS	Classical Gram-Schmidt	klasični Gram-Schmidt
MGS	Modified Gram-Schmidt	modificirani Gram-Schmidt
ICGS	Iterated Classical Gram-Schmidt	iterativni klasični Gram-Schmidt
WSN	Wireless Sensor Networks	brezžična senzorska omrežja
SMP	Symmetric Multiprocessor	simetrični multiprocesorji
PCIe	Peripheral Component Interconnect Express	vodilo za priključitev zunanjih naprav na računalnik
FLOP	Floating-point Operations	enota za število izvršenih operacij s plavajočo vejico
FLOPS	Floating-point Operations Per Second	enota za število izvršenih operacij s plavajočo vejico v sekundi
LMS	Least Mean Squares	metoda najmanjših kvadratov

CDMA	Code Division Multiple Access	multipleksiranje s kodnim deljenjem

Povzetek

Naslov: Paralelizacija Gram-Schmidtovega postopka na sistemu Intel Xeon Phi

Gram-Schmidtov postopek je klasičen postopek za ortonormiranje množice vektorjev v vektorskem prostoru s skalarnim produktom. Obstaja več implementacij Gram-Schmidtovega postopka. Klasičen, modificiran in iterativno klasičen Gram-Schmidtov postopek. Izbrali smo klasičen Gram-Schmidtov postopek. Paralelizacijo klasičnega Gram-Schmidtovega postopka smo realizirali na koprocesorju Intel Xeon Phi. V programskem jeziku C smo s pomočjo knjižnice OpenMP realizirali več funkcij klasičnega Gram-Schmidtovega postopka. Izkazalo se je, da je najboljša funkcija tista, ki enakomerno porazdeli delo med vse niti. Vsaka nit najprej vzporedno izračuna enako število projekcij, nato pa se preostale projekcije izvedejo zaporedno, kjer vse niti vzporedno izvedejo eno projekcijo. Implementacija na Intel Xeon Phi je do trikrat hitrejša kot na dveh Intel Xeon procesorjih. Faktor pohitritve se zvišuje z velikostjo vhodne množice vektorjev.

Ključne besede: paralelizacija, Gram-Schmidt, Intel Xeon Phi.

Abstract

Title: Parallelization of Gram-Schmidt algorithm on Intel Xeon Phi system

The Gram-Schmidt process is a classical process for orthonormalising a set of vectors in inner product space. There are many implementations of the Gram-Schmidt process - classical, modified and iterated classical Gram-Schmidt process. In the thesis we have chosen the classical Gram-Schmidt process. Parallelization of classical Gram-Schmidt process was implemented on Intel Xeon Phi coprocessor. We have implemented many different functions of classical Gram-Schmidt process in the C programming language with a help of OpenMP library. The results show that the best function is the one that evenly distributes work among all threads. First every thread calculates the same amount of projections, then the rest of projections are calculated sequentially, where the threads compute each projection in parallel. The implementation of the same function on the Intel Xeon Phi is up to three times faster than the same implementation on two Intel Xeon processors. The speedup factor is increasing with the size of input vectors.

Keywords: parallelization, Gram-Schmidt, Intel Xeon Phi.

Poglavje 1

Uvod

Gram-Schmidtov postopek je klasičen postopek za ortogonalizacijo množice vektorjev v vektorskem prostoru s skalarnim produktom. Z njim transformiramo bazo izbranega podprostora v ortogonalno bazo istega podprostora, zato se ga uporablja npr. za izračun QR razcepa in reševanje sistemov linearnih enačb.

Gram-Schmidtov postopek se uporablja za ortogonalizacijo kvantno mehaničnih operatorjev in tudi v Lyapunovi stabilnostni analizi sistema delcev [10].

Slučiak in drugi [42] so uporabili porazdeljen QR faktorizacijski algoritem za ortogonalizacijo množice vektorjev v brezžičnem senzorskem omrežju.

Pomembna uporaba Arnoldijevega procesa in s tem posledično Gram-Schmidtovega postopka je v podprostoru Krilova za velike nesimetrične linearne sisteme. GMRES je ena izmed široko uporabljenih metod za reševanje podprostora Krilova [28]. GMRES metoda je doprinesla veliko k oživitvi zanimanja za Gram-Schmidtov postopek.

Gram-Schmidtov postopek se uporablja tudi pri procesiranju signalov [38, 41], izbiri podmnožice, medicini [39], obdelavi slik [34], genetiki, prepoznavi glasov, statistiki [8], modelih za ekonomijo, računalniških omrežjih [29], računalniških iskalnikih, računalniških sistemih [36, 43, 27, 33, 47], simuliranju tokov [12], glasbi [14], fiziki [21, 22, 37], elektrotehniki [25] in seveda

v matematiki: pri Arnoldijevi iteracij [26], pri izračunu Lyapunovih eksponentov [32, 23], pri Choleskyjevi faktorizaciji [31], pri računanju dela spektra velikih skoraj ničelnih matrik [40], pri zvišanju hitrosti konvergence metode najmanjših kvadratov (angl. Least Mean Squares – LMS) v multipleksiranem s kodnim deljenjem (angl. Code Division Multiple Access – CDMA) detektorju [20], pri algoritmih za približke matrik z majhnim rangom [11], pri reševanju problemov nelastnih vrednosti [45].

1.1 Matematični opis Gram-Schmidtovega postopka

Izrek 1.1 *Iz končne množice linearno neodvisnih vektorjev* $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$*, v končno-dimenzionalnem Evklidskem vektorskem prostoru* E*, se lahko sestavi ortogonalno množico neničelnih vektorjev*

$$\boldsymbol{v}_1 = \boldsymbol{u}_1, \quad \boldsymbol{v}_j = \boldsymbol{u}_j - \sum_{i=1}^{j-1} \frac{\langle \boldsymbol{v}_i, \boldsymbol{u}_j \rangle}{\|\boldsymbol{v}_i\|^2} \boldsymbol{v}_i, \quad j \in \{2, 3, \ldots, n\}, \tag{1.1}$$

ki razteza isti podprostor prostora E *kot množica* $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$*.*

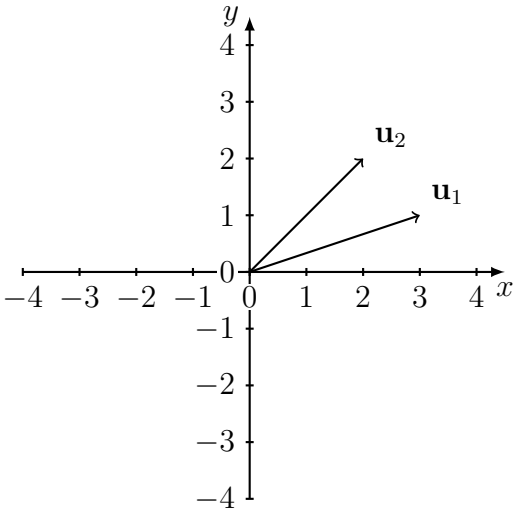
Dokaz (z indukcijo) najdemo v [9]. Če želimo ortonormirano bazo $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, zamenjamo $\mathbf{v}_j$ z $\mathbf{e}_j = \frac{\mathbf{v}_j}{\|\mathbf{v}_j\|}$, pri čemer velja $\|\mathbf{v}_j\| = \sqrt{\langle \mathbf{v}_j, \mathbf{v}_j \rangle}$, za vsak $j \in \{1, 2, \ldots, n\}$. V nadaljevanju privzamemo, da je število vektorjev n enako dimenziji prostora E.

1.2 Grafični prikaz izvajanja Gram-Schmidtovega postopka

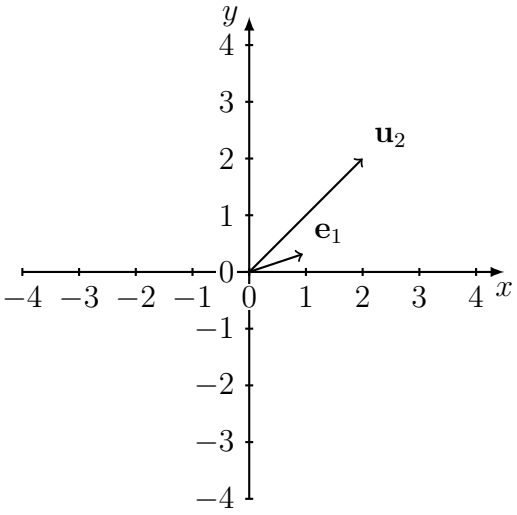
V tem poglavju bomo grafično predstavili Gram-Schmidtov postopek na vektorjih $\mathbf{u}_1 = [3, 1]^T$ in $\mathbf{u}_2 = [2, 2]^T$ v ravnini $\mathbb{R}^2$.

Na spodnjih Slikah 1.1 do 1.6 smo grafično predstavili Gram-Schmidtov postopek za množico vektorjev $\{\mathbf{u}_1, \mathbf{u}_2\}$.

Na Sliki 1.1 sta prikazana začetna vektorja množice $\{\mathbf{u}_1, \mathbf{u}_2\}$. Prvi korak Gram-Schmidtovega postopka je normiranje prvega vektorja. V tem primeru je to vektor $\mathbf{u}_1 = [3, 1]^T$. To prikazuje Slika 1.2, kjer vektor $\mathbf{u}_1$ nadomestimo z vektorjem $\mathbf{e}_1 = [0.948683298, 0.316227766]^T$.



Slika 1.1: Začetna vektorja.



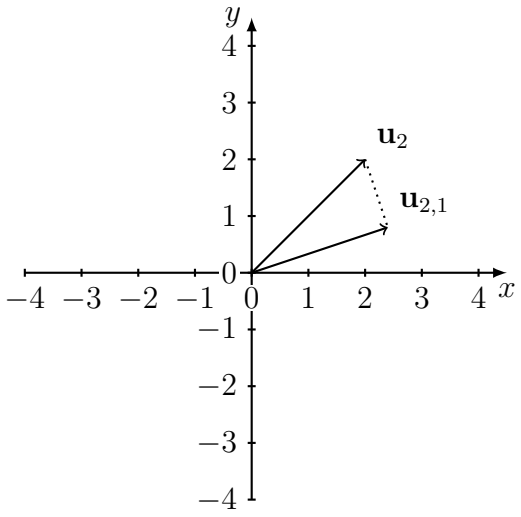
Slika 1.2: Enotski vektor $\mathbf{e}_1$.

Nato naredimo projekcijo vektorja $\mathbf{u}_2$ na enotski vektor $\mathbf{e}_1$, kot prikazuje Slika 1.3. Pri tem dobimo vektor $\mathbf{u}_{2,1} = \overrightarrow{proj_{\mathbf{e}_1}\mathbf{u}_2} = [2.4, 0.8]^T$. Novo dobljeni vektor $\mathbf{u}_{2,1}$ odštejemo od vektorja $\mathbf{u}_2$. Prikaz je na Sliki 1.4.

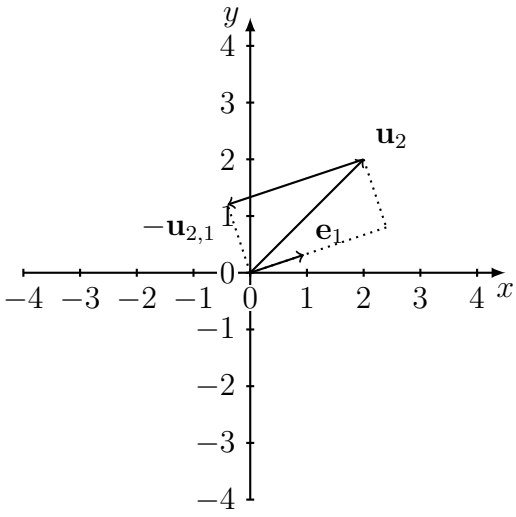
Odštejemo vektor $\mathbf{u}_{2,1}$ od vektorja $\mathbf{u}_2$ in dobimo nov vektor $\mathbf{v}_2 = \mathbf{u}_2 - \mathbf{u}_{2,1} = [-0.4, 1.2]^T$, ki je pravokoten na vektor $\mathbf{e}_1$. Grafični prikaz je na Sliki 1.5. Na koncu Gram-Schmidtovega postopka je potrebno še normirati vektor $\mathbf{u}_2 - \mathbf{u}_{2,1}$. Glej Sliko 1.6. S tem dobimo vektor $\mathbf{e}_2 = [-0.316227766, 0.948683298]^T$. Normirana vektorja $\mathbf{e}_1$ in $\mathbf{e}_2$ sta izhodna vektorja Gram-Schmidtovega postopka.

Tako kot Gram-Schmidtov algoritem vrne različne rezultate za različne vrstne rede vhodnih vektorjev, odvisno na katerem vektorju se bo postopek začel, tako se bo tudi grafični prikaz Gram-Schmidtovega postopka našega primera spremenil. Glej Slike 1.7 do 1.12.

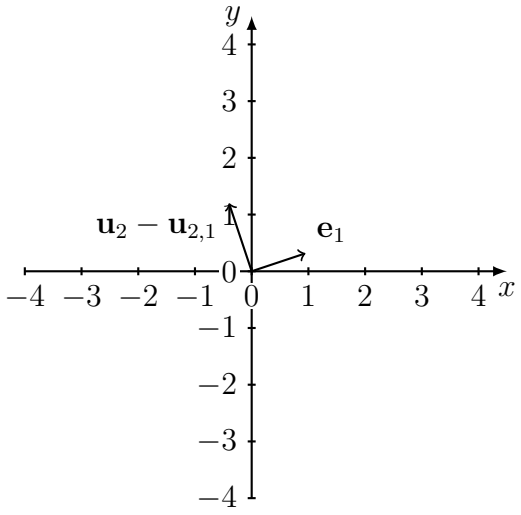
Začetno stanje je prikazano na Sliki 1.7 in je enako stanju na Sliki 1.1. Ker je vhodna množica $\{\mathbf{u}_2, \mathbf{u}_1\}$ bo Gram-Schmidtov postopek najprej normiral



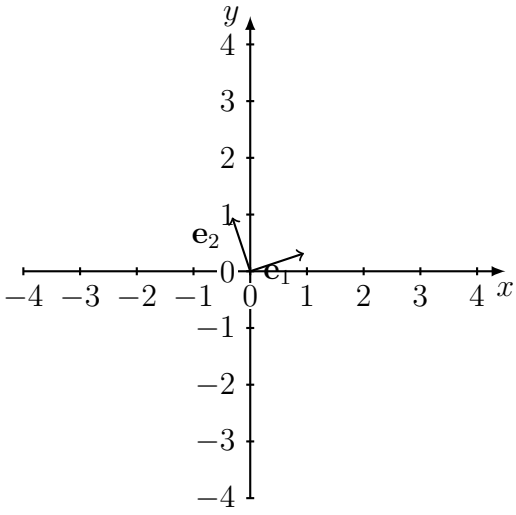
Slika 1.3: Projekcija $\mathbf{u}_{2,1} = [2.4, 0.8]^T$.



Slika 1.4: Vektor $-\mathbf{u}_{2,1}$.



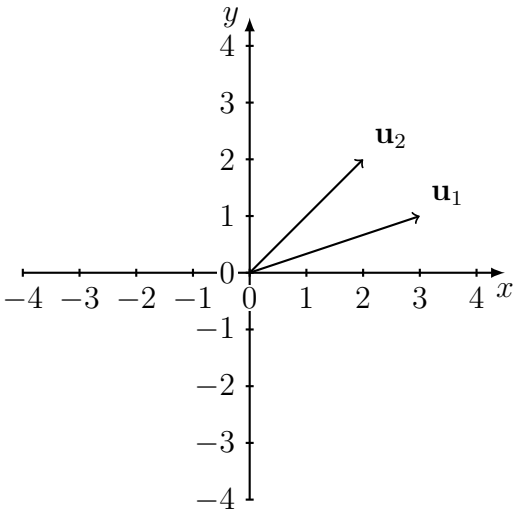
Slika 1.5: Vektor $\mathbf{u}_2 - \mathbf{u}_{2,1} = [-0.4, 1.2]^T$.



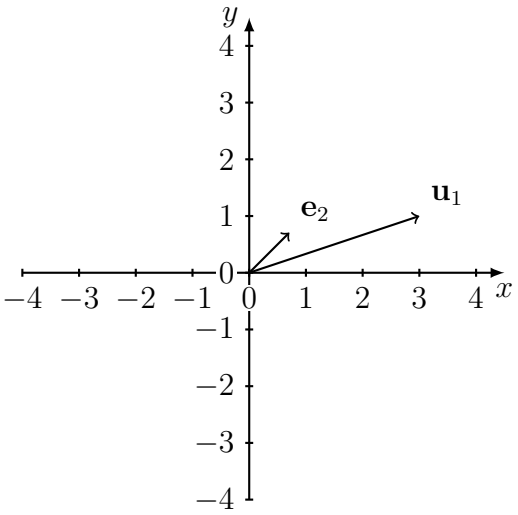
Slika 1.6: Ortonormirana vektorja $\mathbf{e}_1 = [0.9486, 0.3162]^T$ in $\mathbf{e}_2 = [-0.3162, 0.9486]^T$.

vektor $\mathbf{u}_2$. To je prikazano na Sliki 1.8.

Nato se izvede projekcija vektorja $\mathbf{u}_1$ na enotski vektor $\mathbf{e}_2 = [0.707106781, 0.707106781]^T$. Glej Sliko 1.9. Projekcijski vektor $\mathbf{u}_{1,2} = [2, 2]^T$ se prestavi

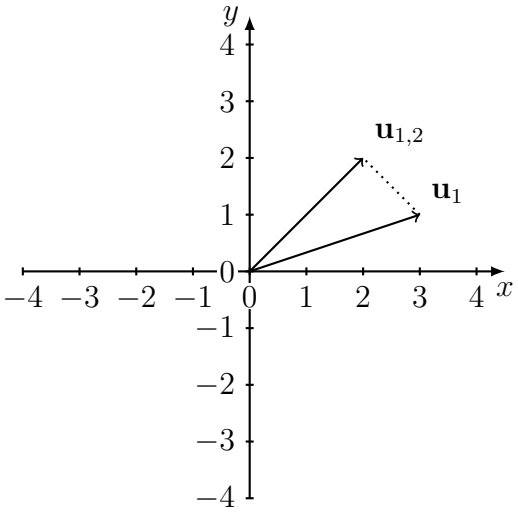


Slika 1.7: Začetna vektorja.

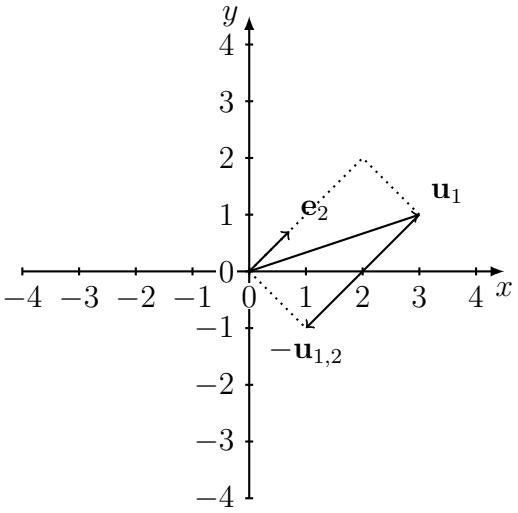


Slika 1.8: Enotski vektor $\mathbf{e}_2$.

na konec vektorja $\mathbf{u}_1$. Glej Sliko 1.10.

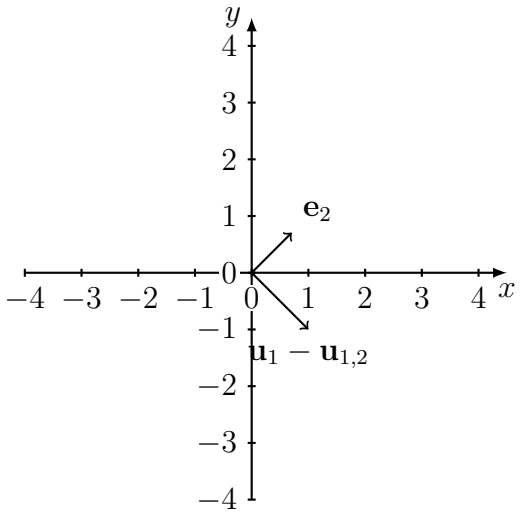


Slika 1.9: Projekcija $\mathbf{u}_{1,2} = [2, 2]^T$.

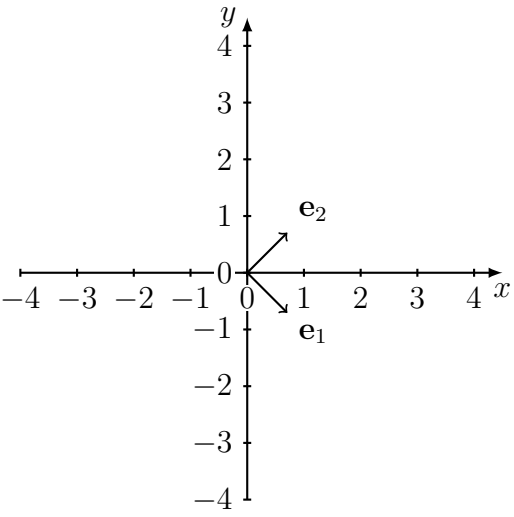


Slika 1.10: Vektor $-\mathbf{u}_{1,2}$.

Vektor $\mathbf{u}_{1,2}$ se odšteje od vektorja $\mathbf{u}_1$. Tako dobimo nov vektor $\mathbf{v}_1 = \mathbf{u}_1 - \mathbf{u}_{1,2} = [1, -1]^T$. Glej Sliko 1.11. Na koncu se izvede še normiranje vektorja $\mathbf{u}_1 - \mathbf{u}_{1,2}$ in dobimo $\mathbf{e}_1 = [0.707106781, -0.707106781]^T$. Prikaz je na Sliki 1.12. S tem smo dobili še drugo izhodno množico Gram-Schmidtovega postopka.



Slika 1.11: Ortogonalen vektor $\mathbf{u}_1 - \mathbf{u}_{1,2} = [1, -1]^T$.



Slika 1.12: Ortonormirana vektorja $\mathbf{e}_1 = [0.7071, -0.7071]^T$ in $\mathbf{e}_2 = [0.7071, 0.7071]^T$.

Iz Slik 1.1 do 1.12 je razvidno, da Gram-Schmidtov postopek vrne različna rezultata za isto vhodno množico. Rezultat je odvisen od vrstnega reda vhodnih vektorjev.

Pri tako preprostem primeru imamo samo dve permutaciji vhodne množice. Če bi imeli n vhodnih vektorjev, bi imeli $n!$ različnih možnosti izhodnih ortogonalnih vektorjev.

1.3 QR razcep

Naj bo A matrika reda $m \times n$ $(m > n)$ in rank(A) = n. Znano je, da lahko matriko A razstavimo kot produkt

$$A = QR,$$

kjer je matrika Q ortogonalna reda $(m \times n)$ in matrika R zgornje trikotna reda $(n \times n)$ [15].

Če ponazorimo na primeru matrike A, iščemo takšen Q in R, da

$$A = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix} = QR.$$

Izkaže se, da je

$$A = \begin{array}{c} \begin{matrix} \mathbf{u}_1 & \mathbf{u}_2 \end{matrix} \\ \begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix} \end{array}$$

$$= \begin{array}{c} \begin{matrix} \mathbf{v}_1 & \mathbf{v}_2 \end{matrix} \\ \begin{bmatrix} 0.94868329805 & -0.31622776601 \\ 0.316227766 & 0.948683298 \end{bmatrix} \end{array} \begin{bmatrix} 3.162277660 & 2.529822128 \\ 0.000000000 & 1.264911064 \end{bmatrix}$$

$$= QR.$$

Če poznamo QR razcep matrike A, potem iz linearnega sistema $A\mathbf{x} = \mathbf{b}$ lahko izračunamo $\mathbf{x}$ [2]. Sistema $A\mathbf{x} = b$ in $R\mathbf{x} = Q^T\mathbf{b}$ sta ekvivalentna, če je le A obrnljiva.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ A^TA\mathbf{x} &= A^T\mathbf{b} \\ (QR)^TQR\mathbf{x} &= (QR)^T\mathbf{b} \\ R^TR\mathbf{x} &= R^TQ^T\mathbf{b}. \end{aligned}$$

Ker je matrika R obrnljiva, dobimo

$$R\mathbf{x} = Q^T\mathbf{b}.$$

Z Gram-Schmidtovim postopkom iz matrike A dobimo matriko Q. S pomočjo matrik A in Q nato izračunamo matriko R. Ko imamo matriki Q in R, samo še poiščemo rešitev sistema.

1.4 Alternativna QR razcepa

Poleg Gram-Schmidtovega postopka obstajajo tudi drugačni postopki za izračun QR razcepa. QR razcep lahko izračunamo tudi s Householderjevimi zrcaljenji ali Givensovimi rotacijami. Vsak ima svoje prednosti in slabosti.

Za matriko A velikosti $m \times n$, zaporedje $n-1$ Householderjevih transformacij spremeni matriko A v zgornje trikotno obliko. Alternativno, zaporedje $m-1$ Givensovih rotacij prav tako spremeni matriko A v zgornje trikotno obliko [16]. Givensove rotacije omogočajo skrivanje komunikacij in imajo pred Householderjevimi zrcaljenji prednost pri vzporedni implementaciji [13].

Z vsemi zgoraj naštetimi postopki lahko izračunamo QR razcep. Ti se razlikujejo v številu operacij in matriki, ki jo vrnejo kot rezultat. Medtem ko Gram-Schmidtov postopek za rezultat vrne ortogonalno matriko Q, Householderjeva zrcaljenja in Givensove rotacije vrnejo zgornje trikotno matriko R. Število operacij za rešitev sistema A$\mathbf{x}$=$\mathbf{b}$ s posameznimi postopki so podane v Tabeli 1.1, kjer m predstavlja število vrstic in n predstavlja število stolpcev vhodne matrike A [5].

1.4.1 Householderjeva zrcaljenja

Definicija Householderjevaga zrcaljenja [3]:

Za vektor $\boldsymbol{w} \in \mathbb{R}^n$, kjer je $\boldsymbol{w} \neq 0$, definiramo

$$P = I - \frac{2}{\boldsymbol{w}^T \boldsymbol{w}} \boldsymbol{w}\boldsymbol{w}^T. \tag{1.2}$$

P je simetrična in ortogonalna matrika, saj je $P = P^T$ in $P^2 = I$. Vsak vektor $\boldsymbol{x} \in \mathbb{R}^n$ lahko zapišemo kot

$$\boldsymbol{x} = \alpha \boldsymbol{w} + \boldsymbol{u}, \tag{1.3}$$

kjer je $\boldsymbol{u} \perp \boldsymbol{w}$. Dobimo

$$P\boldsymbol{x} = -\alpha \boldsymbol{w} + \boldsymbol{u}, \tag{1.4}$$

kar pomeni, da je P zrcaljenje preko hiperravnine, ki je ortogonalna na $\boldsymbol{w}$. Matriko P imenujemo Householderjevo zrcaljenje.

Z množenjem matrike z ustreznimi Householderjevimi zrcaljenji lahko leto spremenimo v zgornjetrikotno obliko:

$$A = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \xrightarrow{\widetilde{P_1}\cdot} \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix} \xrightarrow{\widetilde{P_2}\cdot} \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} = R,$$

kjer $*$ označuje poljubno neničelno vrednost (vsaka $*$ drugo). Pri tem je

$$\widetilde{P}_i = \begin{matrix} \\ i-1 \\ n-i+1 \end{matrix} \begin{matrix} \begin{matrix} i-1 & n-i+1 \end{matrix} \\ \begin{bmatrix} I_{i-1} & 0 \\ 0 & P_i \end{bmatrix} \end{matrix},$$

kjer je n število vrstic matrike.

1.4.2 Givensove rotacije

Definicija 1.1 *Givensova rotacijska matrika $G(i, j, \phi)$ definira k-ti stolpec $G(i, j, \phi)_k$ kot [6]:*

$$G(i,j,\phi)_k := \begin{cases} (\cos\phi)e_i + (\sin\phi)e_j & pri\ k = i \\ -(\sin\phi)e_i + (\cos\phi)e_j & pri\ k = j \\ e_k & drugače \end{cases} \tag{1.5}$$

Matrika

$$\begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

zavrti vektor v pozitivni smeri za kot ϕ [1]. Če izberemo ustrezni kot ϕ, lahko zavrtimo izbrani vektor v vektor, ki ima drugo komponento enako 0.

$$\begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{x^2+y^2} \\ 0 \end{bmatrix}$$

Givensovo rotacijsko matriko $\mathbb{R}^n$, ki zavrti izbrani vektor za kot ϕ v nasprotni smeri urinega kazalca, definiramo kot

$$G(i,j,\phi) = \begin{array}{c} \\ \\ \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccccc} & & & & i & & & & j & & \end{array} \\ \begin{bmatrix} 1 & \dots & & 0 & & \dots & & 0 & & \dots & 0 \\ \vdots & \ddots & & \vdots & & & & \vdots & & & \vdots \\ 0 & \dots & 1 & 0 & & & & 0 & & & 0 \\ 0 & \dots & 0 & \cos\phi & 0 & \dots & 0 & -\sin\phi & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & \sin\phi & 0 & \dots & 0 & \cos\phi & 0 & \dots & 0 \\ 0 & & & 0 & & & & 0 & 1 & \dots & 0 \\ \vdots & & & \vdots & & & & \vdots & & \ddots & \vdots \\ 0 & \dots & & 0 & & \dots & & 0 & & \dots & 1 \end{bmatrix} \end{array},$$

kjer se trigonometrijske funkcije pojavijo v i-tem in j-tem stolpcu.

Primer izračuna:

$$A = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix},$$

kjer $*$ označuje poljubno neničelno vrednost (vsaka $*$ drugo). Ustvarimo rotacijsko matriko G_1, ki bo uničila element v prvem stolpcu in drugi vrstici.

$$G_1(1,2,\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & \\ \sin\phi & \cos\phi & \\ & & 1 \end{bmatrix}$$

Pomnožimo G_1 z leve strani z matriko A.

$$A_1 = G_1 A = \begin{bmatrix} * & * & * \\ 0 & * & * \\ * & * & * \end{bmatrix}$$

Pomnožimo matriko A_1 z rotacijsko matriko $G_2(1,3,\phi')$.

$$A_2 = G_2 A_1 = \begin{bmatrix} \cos\phi' & 0 & -\sin\phi' \\ 0 & 1 & 0 \\ \sin\phi' & 0 & \cos\phi' \end{bmatrix} A_1 = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix}$$

Na koncu pomnožimo matriko A_2 z rotacijsko matriko $G_3(2,3,\phi'')$.

$$A_3 = G_3 A_2 = \begin{bmatrix} 1 & & \\ & \cos\phi'' & -\sin\phi'' \\ & \sin\phi'' & \cos\phi'' \end{bmatrix} A_1 = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} = R$$

1.5 Primerjava Gram-Schmidtovega postopka s Householderjevimi zrcaljenji in Givensovimi rotacijami

Tabela 1.1: Število operacij za rešitev sistema A**x**=**b** s pomočjo QR razcepa.

postopek	število operacij
Gram-Schmidt	$2mn^2$
Householderjeva zrcaljenja	$2mn^2 - 2/3n^3$
Givensove rotacije	$3mn^2 - n^3$

Gram-Schmidtov postopek ima še eno prednost. Gram-Schmidtov postopek je **iterativen**. Kot je razvidno iz definicije Gram-Schmidtovega postopka 1.1, Gram-Schmidtov postopek pri vsaki iteraciji na izhod izpiše en ortonormiran vektor. Householderjeva zrcaljenja in Givensove rotacije kot rezultat vrnejo matriko R, matriko Q pa pridobijo kot produkt refleksij oz. rotacij [10].

Iterativnost in relativna preprostost Gram-Schmidtovega postopka sta glavna razloga, da smo se odločili za implementacijo Gram-Schmidtovega postopka.

1.5.1 Primerjava rezultatov vseh treh QR razcepov

Za primer bomo vzeli ista vektorja kot v Poglavju 1.2. Iz Slike 1.6 je razvidno, da je rezultat Gram-Schmidtovega postopka enak

$$Q = \begin{bmatrix} 0.94868329805 & -0.31622776601 \\ 0.316227766 & 0.948683298 \end{bmatrix}.$$

Prvi stolpec predstavlja vektor $\mathbf{e}_1$ in drugi stolpec predstavlja vektor $\mathbf{e}_2$.

Algoritma za Householderjeva zrcaljenja in Givensove rotacije vrneta zgornje trikotno matriko R. Ker želimo primerjati ortogonalne matrike, moramo za Householderjeva zrcaljenja in Givensove rotacije izračunati še enačbo $Q = AR^{-1}$. Matrika R_h je rezultat Householderjevih zrcaljenj in matrika R_g je rezultat Givensovih rotacij. Matriki Q_h in Q_g vsebujeta ortogonalne vektorje.

$$R_h = \begin{bmatrix} -3.1622776601684 & -2.5298221281347 \\ 0 & -1.2649110640674 \end{bmatrix}$$

$$Q_h = \begin{bmatrix} -0.94868329805051 & 0.31622776601684 \\ -0.31622776601684 & -0.94868329805051 \end{bmatrix}$$

$$R_g = \begin{bmatrix} 3.1622776601684 & 2.5298221281347 \\ 0 & 1.2649110640674 \end{bmatrix}$$

$$Q_g = \begin{bmatrix} 0.94868329805051 & -0.31622776601684 \\ 0.31622776601684 & 0.94868329805051 \end{bmatrix}$$

Givensove rotacije in Householderjeva zrcaljenja so izračunala drugačen rezultat kot Gram-Schmidtov postopek. Obe Q_h in Q_g matriki vsebujeta ortogonalne vektorje. To preverimo tako, da skalarno pomnožimo vektorja med seboj in dobimo 0 za rezultat. Pri Gram-Schmidtovem postopku imamo več možnih pravilnih rezultatov, odvisno od izbire začetnega vektorja. Tudi pri Householderjevih zrcaljenjih in Givensovih rotacijah imamo več možnih začetnih stanj in posledično dobimo mnogo kombinacij za pravilen odgovor Q.

Poglavje 2

Pregled obstoječih rešitev

Gram-Schmidtov postopek je klasičen postopek za določitev QR razcepa in s tem temeljna operacija v linearni algebri. Poleg Gram-Schmidtovega postopka obstajata še dva zelo razširjena algoritma za računanje QR razcepa: Householderjeva zrcaljenja in Givensova rotacija. Čeprav je Gram-Schmidtov postopek počasnejši od Householderjevega zrcaljenja in Givensove rotacije, pa le-ta vrne prve ortogonalne vektorje po le nekaj iteracijah. Gram-Schmidtov postopek v vsaki iteraciji izračuna nov ortogonalen vektor, ki je pravokoten na vse svoje prednike. Na začetku se ortogonalni vektorji računajo hitreje, saj jih je potrebno projecirati na manjšo množico že ortogonalnih vektorjev. Ta iterativna lastnost se uporablja tudi v Arnoldijevi iteraciji. Gram-Schmidtov postopek se uporablja za ortogonalizacijo kvantno mehaničnih operatorjev in tudi v Lyapunovi stabilnostni analizi sistema delcev. Že manj kot 1000 delcev je dovolj, da se Lyapunova stabilnostna analiza računa zelo dolgo. Zato je zelo pomembno, da se pohitri QR razcep. Za pohitritev QR razcepa so napisali paralelni program, optimiziran posebej za arhitekturo na kateri bo tekel [10]. Spoznali so, da je paralelni QR razcep omejen s hitrostjo dostopa do pomnilnika. Za rešitev tega problema so implementirali Gram-Schmidtov postopek, ki se računa po blokih. Algoritem so napisali tako za arhitekturo CPU kot za arhitekturo GPU. Za doseganje dobrih rezultatov priporočajo implementacijo paralelno bločnega algoritma.

Končna hitrost je v veliki meri odvisna tudi od prevajalnika.

Ghodsi in drugi so za izračun linearnega sistema enačb napisali paralelni algoritem, ki temelji na Gram-Schmidtovem postopku [17]. Uporabili so strukturo MIMD, jezik Fortran in knjižnico MPI. Če je velikost problema majhna, ni pohitritve med enim in petimi procesorji. Z večanjem velikosti problema se je izboljševala učinkovitost računanja in večala zakasnitev pri komunikaciji.

Lingen in drugi so primerjali različne verzije Gram-Schmidtovega postopka [30]. Klasičen Gram-Schmidtov postopek (angl. Classical Gram-Schmidt – CGS), modificirani Gram-Schmidtov postopek (angl. Modified Gram-Schmidt – MGS) in iterativni klasičen Gram-Schmidtov postopek (angl. Iterated Classical Gram-Schmidt – ICGS). Primerjali so jih po numerični pravilnosti in stabilnosti, času izvajanja in paralelni učinkovitosti. Noben algoritem se ni računal po blokih. Čeprav je MGS numerično stabilen algoritem, porabi veliko časa za globalno komunikacijo. Numerično nestabilen CGS ima prav toliko operacij s plavajočo vejico kot MGS, vendar zelo malo globalne komunikacije. Numerično stabilnost CGS algoritma so popravili z ICGS algoritmom. ICGS je v nekaterih primerih lahko hitrejši kot MGS, pa čeprav porabi dvakrat toliko operacij s plavajočo vejico. Program se je izvajal na 64 procesorskem Cray T3E in gruči (angl. cluster) osmih delovnih postaj SGI. Za vsakodnevno uporabo je ICGS najboljša izbira, saj je natančnejši in stabilnejši od MGS in mnogo natančnejši in stabilnejši od CGS.

Benjamin Milde in Michael Schnider sta pokazala učinkovito realizacijo algoritma CGS na grafični kartici GeForce 295, z uporabo knjižnice CUDA [35]. Skalarni produkt n-tega elementa ortonormiranega vektorja in n-tega elementa trenutnega vektorja sta odštela od trenutne vrednosti n-tega elementa trenutnega vektorja. Ta princip sta izvedla sočasno za vse n-te elemente preostalih vhodnih vektorjev. Z učinkovito implementacijo in uporabo knjižnice CUDA, sta dobro izkoristila grafične kartice.

Slučiak in drugi so uporabili porazdeljen QR faktorizacijski algoritem za ortogonalizacijo množice vektorjev v brezžičnem senzorskem omrežju [42].

Algoritem izvira iz CGS in uporablja porazdeljen način z uporabo algoritma dinamičnega soglasja. QR faktorizacija je zelo razširjena na področju procesiranja signalov, kjer se večinoma uporablja v povezavi z metodo najmanjših kvadratov. Slučiak in drugi so razvili dva porazdeljena Gram-Schmidtova postopka, ki izračunata matriko Q zaporedno [42]. Predpostavili so, da je brezžično senzorsko omrežje (angl. Wireless Sensor Networks – WSN) statično, povezano in brez napak pri prenosu. Vsako vozlišče izračuna matriko Q glede na svojo lokalno matriko A in nato izvede oceno za matriko R. Simulacije so pokazale, da je konvergenca dinamično soglasnega algoritma CGS močno odvisna od začetne distribucije vrstic po omrežju in topologije omrežja. Uspeli so močno zmanjšati število poslanih sporočil po omrežju, saj le-ti porabijo veliko energije za energetsko omejena omrežja WSN.

Tomás in Hernández sta implementirala ICGS na platformi CUDA [44]. ICGS sta implementirala z knjižnico BLAS2 oziroma knjižnico CUBLAS. Ugotovila sta, da je zmogljivost algoritma omejena s širino pomnilniške povezave. Zato je računanje v dvojni natančnosti trajalo podobno dolgo kot računanje v enojni natančnosti. Z dobro implementacijo jedra (angl. kernel), sta uspela pohitriti računanje za faktor 20, kjer iterativna metoda teče na CPU in zaganja CUBLAS jedra.

Ghodsi in Taeibi-Rahni sta predstavila nov paralelni algoritem, ki temelji na Gram-Schmidtovi ortogonalizaciji [18]. Algoritem najde skoraj točno rešitev tridiagonalnega sistema. Tridiagonalni sistemi se velikokrat pojavijo na področju linearne algebre, delnih diferencialnih enačb, ortogonalnih polinomih in procesiranju signalov. Algoritem razdeli problem na enake particije, kjer vsak procesor dobi svojo particijo. Bistvo algoritma je zmanjšanje komunikacije med procesorji in posledično pohitritev izračuna. Tu bistveno vlogo odigra tridiagonalna matrika, saj za izračun trenutnega ortogonalnega vektorja potrebuje samo dva predhodna vektorja. Vhodno matriko sta razdelila na bloke, vsakemu bloku na koncu odstranila zadnja dva stolpca in jih po vrsti zložila v zadnji prazen blok. Tako se lahko vsi bloki računajo istočasno. Vsak ne zadnji blok pa zelo malo komunicira z zadnjim blokom.

Werneth in ostali so analitično in numerično predstavili uporabo Gram-Schmidtovega postopka na temeljnih funkcijah [46]. Pridobljena ortonormalna množica funkcij močno olajša računanje v kvantni mehaniki. V članku grafično prikažejo ujemanje numerično pridobljenih rezultatov z analitičnimi.

Ker računalniki računajo s približki števil, se lahko zgodi, da rezultati niso točni. Tudi pri Gram-Schmidtovi ortogonalizaciji se lahko zgodi, da matrika Q ni več ortogonalna. Ortogonalnost se lahko popravi tako, da dvakrat projeciramo predhodne Q vektorje na trenutni vhodni vektor. To pa pomeni, da takšen algoritem porabi dvakrat več časa za izračun ortogonalne matrike. Luc Giraud in Julien Langou sta razvila L kriterij, ki realizira kompromis med ohranjanjem ortogonalnosti matrike Q in čim manj podvojenimi projekcijami [19].

Poglavje 3

Razvoj rešitev

V tem poglavju na začetku na kratko opišemo procesorja, na katerih smo izvajali meritve. To sta Intel Xeon procesor in Intel Xeon Phi koprocesor. Nato predstavimo vseh 11 različic vzporednega Gram-Schmidtovega algoritma in razlike med njimi. Na koncu še podrobno opišemo najboljšo rešitev.

3.1 Intel Xeon

Intel Xeon E5-2620 je strežniški procesor, ki je zasnovan na Sandy Bridge mikroarhitekturi. Procesor ima 6 jeder, kjer ima vsako jedro 2 niti. Vsako jedro vsebuje 256-bitno vektorsko enoto. Xeon procesor vsebuje 64 kB L1 predpomnilnika, 256 kB L2 predpomnilnika in 15 MB L3 predpomnilnika. Procesor ima osnovno frekvenco 2 GHz in Intel Turbo Boost tehnologijo, ki zviša frekvenco enega jedra na 2,5 GHz. Računalnik, na katerem so bile opravljene meritve, vsebuje 2 procesorja in 66 GB delovnega pomnilnika.

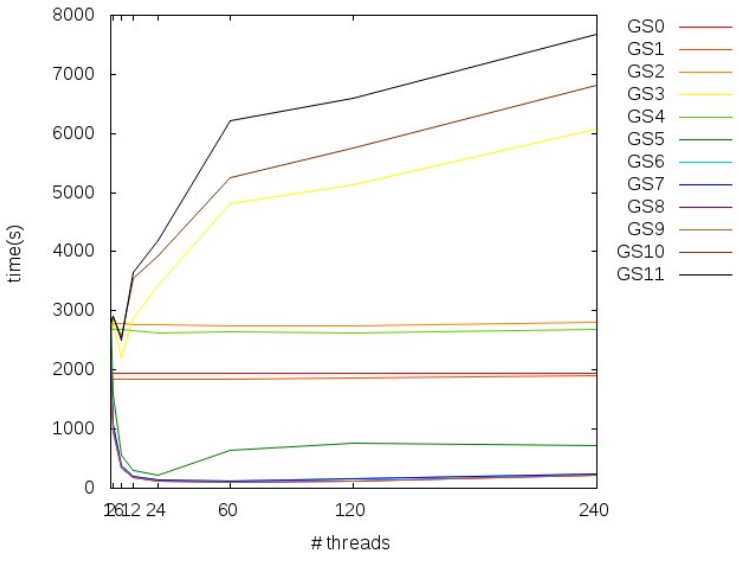
3.2 Intel Xeon Phi

Intel Xeon Phi je koprocesor, osnovan na MIC (Many Integrated Core) arhitekturi in vsebuje 61 jeder z 244 nitmi (4 niti na jedro) in deluje s frekvenco 1 GHz [24]. Procesorska jedra imajo svojega prednika v originalnih Pentium

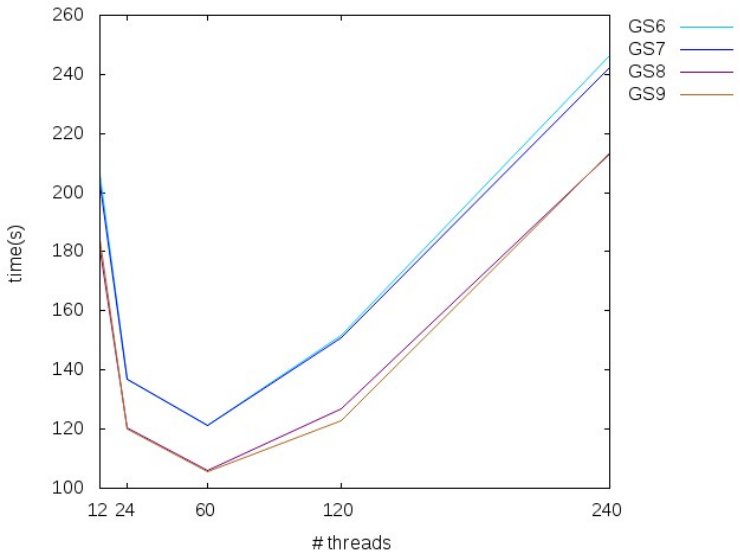
jedrih, katerim je bila dodana 64-bitna podpora, 4 niti na jedro, boljše upravljanje z energijo, 512-bitna vektorska enota. Jedra so med seboj povezana z visokohitrostnim dvosmernim obročem in imajo 512 kB L2 predpomnilnika ter so brez L3 predpomnilnika. Skupaj ima Intel Xeon Phi koprocesor 30 MB L2 predpomnilnika. Na kartici se nahaja še 8 GB GDDR5 delovnega pomnilnika in 8 pomnilniških krmilnikov. Intel Xeon Phi, ki je dostopen preko PCIe vodila, smo uporabili na računalniku iz prejšnjega razdelka.

3.3 Opis rešitev

Kot smo že v uvodu tega poglavja omenili, smo razvili 11 vzporednih različic Gram-Schmidtovega algoritma. Za vsako novo napisano funkcijo smo izmerili hitrost, da smo okvirno izvedeli, če smo s spremembami v kodi pridobili na času. Hitrosti vseh funkcij so predstavljene na Sliki 3.1. Povečava Slike 3.1 je na Sliki 3.2, kjer so prikazane meritve funkcij GS6, GS7, GS8 in GS9. Analiza in primerjava rešitev sledi v Poglavju 4.



Slika 3.1: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 10000 x 10000.



Slika 3.2: Graf časa izvajanj funkcij od GS6 do GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.

Najprej smo napisali funkcijo GS0, ki nam je služila za preverjanje rezultatov in za primerjavo časa. Nato smo napisali funkciji GS1 in GS2, ki sta prvi funkciji napisani vzporedno s pomočjo OpenMP knjižnice. Zaradi slabih rezultatov funkcij GS1 in GS2 smo razvili funkciji GS3 in GS4, na katerih smo se še vedno učili ter posledično uporabljali premalo oz. preveč OpenMP anotacij. Vse funkcije smo na začetku izmerili na največ 5000 vektorjih. Kronološko je sledila funkcija GS5, ki je prvič pravilno uporabila vse OpenMP anotacije, kar se vidi tudi na Sliki 3.1. Nato smo napisali sestrski funkciji GS6 in GS7, ki sta še izboljšali hitrost v primerjavi s funkcijo GS5. Vse zgoraj omenjene funkcije delujejo nad dvema tabelama, in sicer iz vhodne samo berejo podatke v izhodno pa zapišejo rezultat. Za še večjo pohitritev smo realizirali funkcijo GS8, ki deluje nad eno samo tabelo. Za konec smo napisali še funkcije GS9, GS10 in GS11, ki imajo osnovo v funkciji GS8 in se razlikujejo samo v OpenMP anotacijah in razbitjih zank.

3.3.1 Funkcija GS0

Na začetku smo napisali zaporedno funkcijo Gram-Schmidtovega algoritma, ki smo jo poimenovali GS0 in je predstavljena v Prilogi A.1. Funkcijo GS0 smo napisali z namenom preverjanja pravilnosti rezultatov in tudi z namenom primerjanja hitrosti glede na vzporedne funkcije.

Vse funkcije, ki računajo Gram-Schmidtov algoritem, na začetku prepišejo vhodno matriko, ki jo preberejo na standardnem vhodu, v izhodno matriko. V tej izhodni matriki se po koncu izvajanja funkcije nahaja končna rešitev.

3.3.2 Funkcija GS1

Funkcija GS1 je bila prva vzporedno napisana funkcija. Njena koda je v Prilogi A.2. Prepis vhodne matrike v izhodno matriko se izvede vzporedno. Nato smo zaporedno izračunali skalarni produkt prvega vhodnega vektorja samega s seboj, ki smo ga potrebovali za vzporedno računanje prvega enotskega vektorja. Vsak element prvega vektorja smo vzporedno pomnožili z $1/\sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$ in tako dobili prvi enotski vektor.

Nato smo z uporabo OpenMP anotacije `#pragma omp parallel for ordered`, zaporedno izračunali še ostale ortogonalne vektorje. Anotacija `parallel for` prevajalniku pove, da se bo zanka izvajala vzporedno. Anotacija `order` pa pove, da se bo znotraj vzporednega bloka izvedla zaporedna koda.

```
#pragma omp parallel for ordered
for i=1 to n:
  //od tu naprej izvajaj kot, da je koda zaporedna
  {
    ∀v_j kjer je j<i naredi:
      dot ← ⟨v_i, v_j⟩
      v_i ← v_i − dot ∗ v_j
    for i=1 to n:
      v_i ← 1/‖v_i‖ v_i
```

```
}
```

Psevdokoda 3.1: Projekcija ortogonalnih vektorjev na trenutni vektor $\mathbf{v}_i$ v funkciji GS1.

Za vsako iteracijo zanke se izvedejo vse projekcije ortonormiranih vektorjev na trenutni vektor $\mathbf{v}_i$ zaporedno. Trenutni vektor normiramo, nato se zanka ponovi.

3.3.3 Funkcija GS2

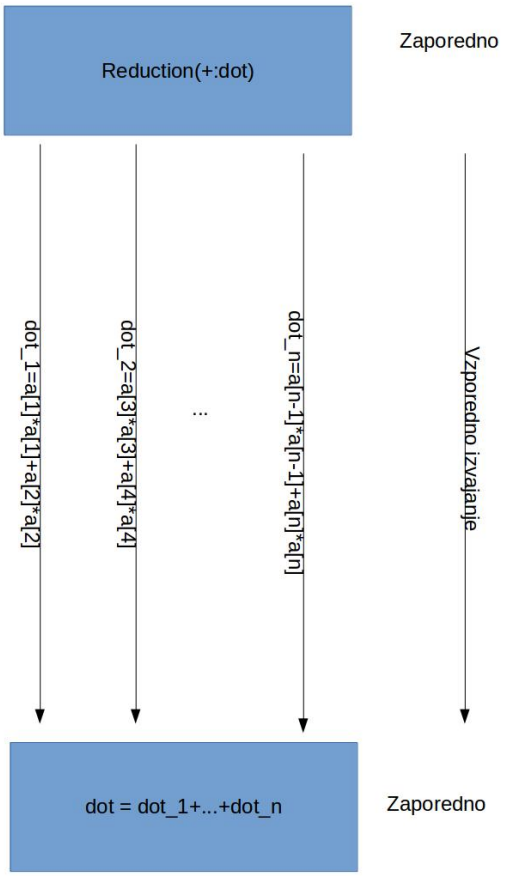
Funkcija GS2, predstavljena v Prilogi A.3, je izpeljana iz funkcije GS1. Spremenjen je izračun skalarnega produkta prvega vhodnega vektorja samega s seboj. Sedaj se skalarni produkt izvede vzporedno s pomočjo `reduction` anotacije iz knjižnice OpenMP.

Z anotacijo `reduction` označimo številsko spremenljivko na kateri izvedemo eno izmed asociativnih matematičnih operacij. Z anotacijo `reduction` vsaka nit izvede asociativno operacijo nad svojo lokalno spremenljivko. Te lokalne spremenljivke se ob koncu anotacije `parallel` seštejejo v končno globalno spremenljivko. Glej Sliko 3.3.

Kot v funkciji GS1, se v funkciji GS2 izvede vzporedno normiranje prvega vektorja. Nato se izračun vsakega novega ortogonalnega vektorja izvede zaporedno z OpenMP anotacijo `ordered`.

Ker je naš glavni cilj pohitriti Gram-Schmidtov postopek za velikost matrike 10000, smo pohitrili izračun skalarnega produkta in projekcije vseh ortogonalnih vektorjev na trenutnega. Glej Psevdokodo 3.2. To smo realizirali z istimi anotacijami, kot pri skalarnem produktu prvega vektorja samega s seboj.

```
#pragma omp parallel for ordered
for i=1 to n:
  //od tu naprej izvajaj kot, da je koda zaporedna
  {
```



Slika 3.3: Slika izvajanja OpenMP reduction anotacije.

```
    for j=0 to j<i:
        #pragma omp parallel for reduction(dot)
        dot ← ⟨v_i, v_j⟩
        #pragma omp parallel for
        v_i ← v_i − dot ∗ v_j
}
#pragma omp parallel for
for i=0 to n:
```

$\mathbf{v}_i \leftarrow \frac{1}{\|\mathbf{v}_i\|}\mathbf{v}_i$

Psevdokoda 3.2: Psevdokoda GS2.

Normiranje trenutnega vektorja se izvede vzporedno v dveh korakih. Najprej vzporedno izračunamo skalarni produkt trenutnega vektorja samega s seboj, nato pa v drugem koraku vzporedno pomnožimo vse elemente vektorja z $1/\sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$.

Vse to je bilo narejeno z namenom, da bi pridobili na hitrosti. Ker je bil naš glavni cilj razviti algoritem za velikost vektorjev 10000, smo realizirali elementarne operacije kot so skalarni produkt, projekcija in normiranje, vzporedno. Mislili smo, da bo to precej pohitrilo celoten algoritem.

Najprej smo procesorju dejali, da se bodo ortogonalni vektorji projecirali vzporedno. Nato so se morale vse že ustvarjene vzporedne niti izvesti zaporedno ena za drugo. Znotraj teh niti smo klicali nov vzporedni konstrukt, ki je zahteval nove niti za vzporedno računanje skalarnega produkta in projekcije. Ker pa je že zunanja zanka porabila vse niti, jih skalarni produkt in projekcija nista mogla izkoristiti. OpenMP knjižnica je za vsako iteracijo zanke poskušala pridobiti nove niti za vzporedno izvajanje skalarnega produkta in projekcije.

Glavni razlog za počasnost je v uporabi OpenMP anotacije `ordered`, s katero smo napisali zaporedno kodo. S tem smo v funkcijo GS2 vnesli samo dodatno komunikacijo, ustvarjanje in brisanje niti. To je razlog, zakaj je funkcija GS2 mnogo slabša od funkcije GS1.

3.3.4 Funkcija GS3

Funkcija GS3 je izpeljana iz funkcije GS2 in ima enak izračun prvega ortogonalnega vektorja. Nato smo odstranili zaporedno OpenMP anotacijo za izvajanje ortonormiranj vseh preostalih vektorjev, saj je to enako, kot če anotacije ni. Glej Psevdokodo 3.3.

```
#pragma omp parallel for ordered
```

```
for i=1 to n:
  //od tu naprej izvajaj kot, da je koda zaporedna
  {
    for j=0 to j<i:
      #pragma omp parallel for reduction(dot)
      dot ← ⟨v_i, v_j⟩
      #pragma omp parallel for
      v_i ← v_i − dot ∗ v_j
  }
  #pragma omp parallelfor
  for i=0 to n:
    v_i ← 1/‖v_i‖ v_i
```

Psevdokoda 3.3: Psevdokoda GS3.

Projekcije in normiranje na posamezen vektor se še vedno izvedejo vzporedno. Sprememba med GS3 in GS2 je tudi v tem, kako potekajo projekcije. V zgornjih dveh funkcijah smo projecirali vse ortogonalne vektorje na trenutni vektor, sedaj pa trenutni ortogonalni vektor najprej projeciramo na vse preostale neortogonalne vektorje.

3.3.5 Funkcija GS4

Funkcija GS2 ima še eno izpeljanko. To je funkcija GS4. V GS2 so se posamične projekcije na trenutni vektor izvajale zaporedno. Tukaj pa smo hoteli projekcije pospešiti, zato smo dodali anotacijo `parallel` na zanko, ki izvede projekcije ortogonalnih vektorjev na trenutni vektor. Glej Psevdokodo 3.4.

```
#pragma omp parallel for ordered
for i=1 to n:
  //od tu naprej izvajaj kot, da je koda zaporedna
  {
```

```
    #pragma omp parallel for
    for j=0 to j<i:
      #pragma omp parallel for reduction(dot)
      dot ← ⟨v_i, v_j⟩
      #pragma omp parallel for
      v_i ← v_i − dot ∗ v_j

    #pragma omp parallel for
    for i=0 to n:
      v_i ← 1/‖v_i‖ v_i
}
```

Psevdokoda 3.4: Psevdokoda GS4.

To smemo narediti, saj je rezultat enak, če na izhodni vektor najprej projeciramo prvi ortonormiran vektor ali pa n-tega. Ker je preostanek kode enak kot v funkciji GS2 in zunanja zanka še vedno uporablja anotacijo `ordered`, je pričakovano hitrost GS4 funkcije le za odtenek boljša od funkcije GS2.

3.3.6 Funkcija GS5

Tudi funkcija GS3 je dobila svojo izpeljanko oziroma nadgradnjo v funkciji GS5. Bolje rečeno, funkcija GS5 je mešanica funkcije GS3 in GS4. Glej Psevdokodi 3.3 in 3.4. Od funkcije GS3 je podedovala projekcijo prvega ortogonalnega vektorja na vse ostale neortogonalne vektorje in šele na to projeciramo drug ortogonalen vektor. Po funkciji GS4 pa je podedovala vzporedno projeciranje ortogonalnih vektorjev na trenutni vektor. Ima pa funkcija GS5 dodano lastnost, da se pri vsaki projekciji hkrati v isti zanki izračuna še skalarni produkt tega vektorja samega s seboj. Rezultat, ki ga potrebujemo za normiranje trenutnega vektorja, pa je pravilen takrat, ko se izvede projekcija na trenutni vektor. Ta skalarni produkt se lahko izvede med prvimi vzporednimi izvajanji ali pa med zadnjimi. Zato potrebujemo pogojni stavek, ki shrani pravilni skalarni produkt, katerega potem uporabimo pri normiranju

trenutnega vektorja. Glej Psevdoko 3.5. To smo naredili z razlogom, da smo se znebili ponovnega računanja skalarnega produkta za novi i-ti ortogonalen vektor. Hkrati smo s tem izboljšali dostop do pomnilnika, saj se `dot1` računa istočasno ob izvajanju projekcije, kar se pozna tudi na času izvajanja.

```
for i=1 to n:
  #pragma omp parallel for
  for j=n-1 to j>=i:
    #pragma omp parallel for reduction(dot)
    dot ← ⟨v_i, v_j⟩
    #pragma omp parallel for
    {
      v_i ← v_i − dot ∗ v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
    shrani dot1, ko je i==j
  #pragma omp parallel for
  for i=0 to n:
    v_i ← 1/‖v_i‖ v_i
```

Psevdokoda 3.5: Psevdokoda GS5.

3.3.7 Funkcija GS6

Funkcija GS6 je izpeljanka funkcije GS5. Glej Psevdoko 3.6. Funkcija GS6 izračuna, koliko projekcij vektorjev bo izvedla vzporedno, glede na število niti, ki jih ima program na voljo. Če ima program na voljo k-niti, se bo prvih m-projekcij izvedlo vzporedno, kjer se vsaka projekcija računa zaporedno, ostale projekcije pa se izvedejo zaporedno, kjer se skalarni produkt in sama projekcija izvedejo vzporedno. m je največji večkratnik števila niti, ki je manjši ali enak trenutnemu številu ortogonalnih vektorjev. To je bilo

narejeno z namenom, da se bodo vse vzporedne projekcije končale istočasno in ne bo treba čakati ene niti, ki je dobila za izračun eno projekcijo več kot ostale. Prav tako se normiranje trenutnega vektorja izvede najprej za m-elementov vzporedno, preostanek pa se izračuna zaporedno. Enako velja tudi za normiranje prvega vektorja. S funkcijo GS6 smo hoteli enakomerno zasesti niti.

```
m ← (velikost_vektorja/st_niti) * st_niti
#pragma omp parallel for
for j=n-1 to j>=m:
   dot ← dot + ⟨v_0, v_0⟩
for j=m-1 to j>=0:
   dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n-1 to j>=m:
   ||v_0||
for j=m-1 to j>=0:
   ||v_0||
for i=1 to n:
   m ← i + (velikost_vektorja - i)%st_niti
   #pragma omp parallel for
   for j=n-1 to j>=m:
      dot ← ⟨v_i, v_j⟩
      v_i ← v_i - dot * v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
      shrani dot1, ko je i == j
   for j=m-1 to j>=i:
      #pragma omp parallel for reduction(dot)
      dot ← ⟨v_i, v_j⟩
      #pragma omp parallel for
```

```
    {
      v_i ← v_i − dot ∗ v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
    shrani dot1, ko je i == j
  #pragma omp parallel for
  for j=n−1 to j>=m:
    v_j ← 1/‖v_j‖ v_j
  for j=m−1 to j>=i:
    v_j ← 1/‖v_j‖ v_j
```

Psevdokoda 3.6: Psevdokoda GS6.

3.3.8 Funkcija GS7

V kronološkem zaporedju sledi izpeljanka funkcije GS6. To je funkcija GS7. Edina razlika med tema dvema funkcijama je v definiciji vhodne in izhodne matrike. Medtem, ko je v GS6 vhodna in izhodna matrika navadna dvodimenzionalna tabela, smo tema dvema matrikama oz. tabelama v GS7 dodali `__attribute__((aligned(64)))`. Glej Psevdoko 3.7. S tem dodatkom smo hoteli zagotoviti, da so podatki poravnani, saj vemo, da procesor večinoma časa dostopa do zaporednih podatkov, in s tem pohitriti izračun Gram-Schmidtovega postopka. Glede na to majhno modifikacijo in enake čase izvajanja med GS6 in GS7 lahko sklepamo, da so tabele v GS6 in vseh ostalih funkcijah prav tako poravnane.

```
//inicializacija poravnane vhodne in izhodne tabele
m ← (velikost_vektorja/st_niti) ∗ st_niti
#pragma omp parallel for
for j=n−1 to j>=m:
  dot ← dot + ⟨v_0, v_0⟩
```

```
for j=m-1 to j>=0:
  dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n-1 to j>=m:
  ||v_0||
for j=m-1 to j>=0:
  ||v_0||
for i=1 to n:
  m ← i + (velikost_vektorja − i)%st_niti
  #pragma omp parallel for
  for j=n-1 to j>=m:
    dot ← ⟨v_i, v_j⟩
    v_i ← v_i − dot * v_j
    //in hkrati izracunaj trenutni dot
    dot1 ← ⟨v_i, v_i⟩
    shrani dot1, ko je i == j
  for j=m-1 to j>=i:
    #pragma omp parallel for reduction(dot)
    dot ← ⟨v_i, v_j⟩
    #pragma omp parallel for
    {
      v_i ← v_i − dot * v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
    shrani dot1, ko je i == j
  #pragma omp parallel for
  for j=n-1 to j>=m:
    v_j ← 1/||v_j|| v_j
  for j=m-1 to j>=i:
```

$\mathbf{v}_j \leftarrow \frac{1}{\|\mathbf{v}_j\|}\mathbf{v}_j$

Psevdokoda 3.7: Psevdokoda GS7.

3.3.9 Funkcija GS8

Funkcija GS8 je izpeljanka iz funkcije GS7. V GS8 smo se znebili ene tabele. Glej Psevdokodo 3.8. Sedaj se bere in piše v eno tabelo. S tem se je naš algoritem spremenil v MGS, saj funkcija za skalarni produkt vzame vhodni vektor in modificiran ortogonalen vektor. S tem korakom smo hoteli izboljšati lokalno dostopnost spremenljivk. Hitrost izračuna se je v povprečju izboljšala za 13 % glede na funkcijo GS7. Podroben opis te funkcije pa se nahaja v Poglavju 3.4.

```
//inicializacija poravnane vhodne ~~in izhodne~~ tabele
m ← (velikost_vektorja/st_niti) * st_niti
#pragma omp parallel for
for j=n-1 to j>=m:
  dot ← dot + ⟨v_0, v_0⟩
for j=m-1 to j>=0:
  dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n-1 to j>=m:
  ||v_0||
for j=m-1 to j>=0:
  ||v_0||
for i=1 to n:
  m ← i + (velikost_vektorja - i)%st_niti
  #pragma omp parallel for
  for j=n-1 to j>=m:
    dot ← ⟨v_i, v_j⟩
    v_i ← v_i - dot * v_j
```

```
    //in hkrati izracunaj trenutni dot
    dot1 ← ⟨v_i, v_i⟩
    shrani dot1, ko je i == j
  for j=m−1 to j>=i:
    #pragma omp parallel for reduction(dot)
    dot ← ⟨v_i, v_j⟩
    #pragma omp parallel for
    {
      v_i ← v_i − dot ∗ v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
    shrani dot1, ko je i == j
  #pragma omp parallel for
  for j=n−1 to j>=m:
    v_j ← 1/‖v_j‖ v_j
  for j=m−1 to j>=i:
    v_j ← 1/‖v_j‖ v_j
```

Psevdokoda 3.8: Psevdokoda GS8.

3.3.10 Funkcija GS9

Funkcija GS9 kronološko sledi funkciji GS8. Glej Psevdokodo 3.9. Tu je bil postopek skoraj ravno obraten kot pri prehodu iz funkcije GS5 v GS6. V funkciji GS9 se projekcije najprej izvedejo vzporedno za večkratnik števila niti, nato se preostanek projekcij izvede zaporedno, kjer se skalarni produkt in same projekcije izvedejo vzporedno. Začetno normiranje, začetni skalarni produkti in končna normiranja se izvedejo v eni zanki z OpenMP anotacijo za vzporedno izvajanje. Mnenja smo, da začetno normiranje in končno normiranje ne porabita toliko časa, ker delujeta samo nad enim vektorjem, kot skalarni produkti in projekcije pri projeciranju ortogonalnih vektorjev na

trenutni vektor. S tem smo želeli zmanjšati in poenostaviti Kodo A.5.

```
//inicializacija poravnane vhodne tabele
m ← (velikost_vektorja/st_niti) * st_niti
#pragma omp parallel for
for j=n−1 to j>=0:
  dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n−1 to j>=0:
  ||v_0||
for i=1 to n:
  m ← i + (velikost_vektorja − i)%st_niti
  #pragma omp parallel for
  for j=n−1 to j>=m:
    dot ← ⟨v_i, v_j⟩
    v_i ← v_i − dot * v_j
    //in hkrati izracunaj trenutni dot
    dot1 ← ⟨v_i, v_i⟩
    shrani dot1, ko je i == j
  for j=m−1 to j>=i:
    #pragma omp parallel for reduction(dot)
    dot ← ⟨v_i, v_j⟩
    #pragma omp parallel for
    {
      v_i ← v_i − dot * v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
    shrani dot1, ko je i == j
  #pragma omp parallel for
  for j=n−1 to j>=i:
```

$\mathbf{v}_j \leftarrow \frac{1}{\|\mathbf{v}_j\|}\mathbf{v}_j$

Psevdokoda 3.9: Psevdokoda GS9.

3.3.11 Funkcija GS10

Funkcija GS10 prav tako izhaja iz funkcije GS8. Glej Psevdokodo 3.10. Skalarni produkt in normiranje prvega vektorja ter normiranje n-tega izhodnega vektorja se izvede ločeno. Prvih m-elementov se izračuna vzporedno, ostanek pa zaporedno. V funkciji GS10 se projekcije izvedejo zaporedno. V tej funkciji smo hoteli preveriti ali je zaporedno izvajanje projekcij, znotraj katerih se vzporedno izračuna skalarni produkt in projekcija, hitrejše od vzporednih projekcij. Kot lahko vidimo iz Slike 3.1, je vzporedno apliciranje projekcij mnogo hitrejše kot zaporedno izvajanje vzporednih projekcij.

```
//inicializacija poravnane vhodne tabele
m ← (velikost_vektorja/st_niti) * st_niti
#pragma omp parallel for
for j=n-1 to j>=m:
   dot ← dot + ⟨v_0, v_0⟩
for j=m-1 to j>=0:
   dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n-1 to j>=m:
   ||v_0||
for j=m-1 to j>=0:
   ||v_0||
for i=1 to n:
   for j=n-1 to j>=i:
      #pragma omp parallel for reduction(dot)
      dot ← ⟨v_i, v_j⟩
      #pragma omp parallel for
```

```
    {
      v_i ← v_i − dot ∗ v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
  #pragma omp parallel for
  for j=n−1 to j>=m:
    v_j ← 1/‖v_j‖ v_j
  for j=m−1 to j>=i :
    v_j ← 1/‖v_j‖ v_j
```

Psevdokoda 3.10: Psevdokoda GS10.

3.3.12 Funkcija GS11

Zadnja realizirana funkcija GS11 ima svojega prednika v funkciji GS8. Glej Psevdokodo 3.11. Vsa normiranja in skalarni produkti se izvedejo vzporedno z eno OpenMP anotacijo. Vse projekcije se izvedejo zaporedno, znotraj katerih se skalarni produkti in odštevanja izvajajo vzporedno. Tudi ta funkcija ne doseže želene pohitritve, saj ima enako napako kot funkcija GS10. Kot smo ugotovili že pri funkciji GS10, je za hitrost nujno potrebno vzporedno izvajati projekcije. V tej funkciji pa smo želeli videti ali lahko zaporedno izvajanje projekcij, ki se izvedejo vzporedno, prehiti našo najboljšo funkcijo, GS8.

```
//inicializacija poravnane vhodne tabele
#pragma omp parallel for
for j=n−1 to j>=0:
  dot ← dot + ⟨v_0, v_0⟩
#pragma omp parallel for
for j=n−1 to j>=0:
  ‖v_0‖
```

```
for i=1 to n:
  for j=n-1 to j>=i:
    #pragma omp parallel for reduction(dot)
    dot ← ⟨v_i, v_j⟩
    #pragma omp parallel for
    {
      v_i ← v_i − dot ∗ v_j
      //in hkrati izracunaj trenutni dot
      dot1 ← ⟨v_i, v_i⟩
    }
  for j=n-1 to j>=i:
    v_j ← 1/‖v_j‖ v_j
```

Psevdokoda 3.11: Psevdokoda GS11.

3.4 Podroben opis funkcije GS8

Kot smo omenili že v Poglavju 3.3, v funkciji GS8 uporabljamo samo eno tabelo za računanje vektorjev Gram-Schmidtovega postopka. Funkcija najprej izračuna širino tabele, nad katero se bo izvedla vzporedna koda. Glej Kodo 3.12. Konstanta `NUM_OF_VEC_AND_COMP` predstavlja število elementov vektorjev in hkrati tudi število vektorjev.

```
195 void GS8(double inputVectors[][NUM_OF_VEC_AND_COMP], int numOfThreads)
196 {
197   int firstIterations = (NUM_OF_VEC_AND_COMP/numOfThreads)*numOfThreads;
```

Koda 3.12: Izračun dolžine tabele, ki se bo izvedla vzporedno.

Nato vzporedno izračunamo skalarni produkt za to dolžino tabele. Glej Kodo 3.13. OpenMP `reduction` anotacija na koncu sešteje privatne spremenljivke `dot2` vseh niti skupaj v globalno spremenljivko `dot2`. Anotacija `private` pa poskrbi, da ima vsaka nit svojo instanco spremenljivke i. Pre-

ostanek skalarnega produkta izračunamo zaporedno. Dolžina, ki se izvede zaporedno, je vedno manjša kot število niti `numOfThreads`.

```
199 #pragma omp parallel for default(none) shared(inputVectors,firstIterations↩
        ) private(i) reduction(+:dot2)
200 for(i=0; i<firstIterations; i++){
201     dot2 += inputVectors[0][i]*inputVectors[0][i];
202 }
203
204 for(i=firstIterations; i<NUM_OF_VEC_AND_COMP; i++) {
205   dot2 += inputVectors[0][i]*inputVectors[0][i];
206 }
```

Koda 3.13: Izračun skalarnega produkta.

Nato se izračuna vrednost $1/\sqrt{\texttt{dot2}}$, s katero se množi vsak element prvega vektorja. To se izvede na enak način kot vsota `dot2`. Na začetku se vsa deljenja izvedejo vzporedno, ostanek pa se izračuna zaporedno. Na ta način je prvi vektor normiran.

Po prvem normiranem vektorju se algoritem zaporedno sprehodi čez vse ostale vektorje. Na vsakem koraku izračuna optimalno širino tabele, nad katero se bo pognala vzporedna Koda 3.14.

```
219 for(k=1; k<NUM_OF_VEC_AND_COMP; k++){
220   double dot1 = 0;
221
222   int firstIt = NUM_OF_VEC_AND_COMP;
223   if(NUM_OF_VEC_AND_COMP-k>=numOfThreads) {
224     firstIt = k+((NUM_OF_VEC_AND_COMP-k)%numOfThreads);
225   }
```

Koda 3.14: Izračun dolžine tabele, za vzporedno procesiranje.

Z znanim številom vektorjev, ki se bodo izvedli vzporedno, lahko začnemo z izračunom projekcij ortogonalnega vektorja z indeksom k na preostale ne ortogonalne vektorje, Koda 3.15.

```
227 #pragma omp parallel for default(none) shared(inputVectors, dot1, firstIt)↩
        private(i,j) firstprivate(k)
```

```
228 for(i=NUM_OF_VEC_AND_COMP-1; i>=firstIt; i--) {
229   //izracun skalarnega produtka k-tega in i-tega vektorja
230
231   double dotTmp = 0;
232   for(j=0; j<NUM_OF_VEC_AND_COMP; j++){
233     inputVectors[i][j] -= dot*inputVectors[k-1][j];
234     dotTmp += inputVectors[i][j]*inputVectors[i][j];
235   }
236   //belezenje skalarnega produkta i-tega vektorja
237   if(i==k) {
238     dot1 = dotTmp;
239   }
240 }
```

Koda 3.15: Projekcija ortogonalnega vektorja na neortogonalne vektorje.

Hkrati tudi računamo skalarni produkt `dotTmp` potencialno novega ortogonalnega vektorja. Skalarni produkt novega ortogonalnega vektorja moramo pridobiti preko pogojnega stavka `if(i==k)`. Ker se ta zanka izvaja vzporedno, ne moremo vedeti, kdaj se bo, če se bo, izvedla projekcija k-tega ortogonalnega vektorja na i-tega. Izpolnjen pogoj pomeni, da so se na i-ti vektor projecirali že vsi ortogonalni vektorji, kar pomeni, da je sedaj ortogonalen na vse vektorje, ki so v tabeli nad njim.

Če število neortogonalnih vektorjev ni deljivo s številom niti, se naslednje projekcije posameznega k-tega vektorja izvajajo zaporedno na i-ti vektor. Sam skalarni produkt in projekcija i-tega vektorja s k-tim se izvede vzporedno, kot prikazuje Koda 3.16.

```
248 for(i=firstIt-1; i>=k; i--) {
249   double dot = 0;
250   #pragma omp parallel for private(j) reduction(+:dot)
251   for(j=0; j<NUM_OF_VEC_AND_COMP; j++){
252     dot += inputVectors[i][j]*inputVectors[k-1][j];
253   }
254
255   dot1 = 0;
256   #pragma omp parallel for shared(dot) private(j) reduction(+:dotTmp)
257   for(j=0; j<NUM_OF_VEC_AND_COMP; j++){
258       inputVectors[i][j] -= dot*inputVectors[k-1][j];
259       dot1 += inputVectors[i][j]*inputVectors[i][j];
260   }
```

```
261 }
```

Koda 3.16: Preostanek projekcij ortogonalnega vektorja.

Na koncu se izvede samo še normiranje k-tega vektorja, na enak način kot Koda v 3.13. Celotna funkcija GS8 je zapisana v Dodatku A.4.

Poglavje 4

Primerjava rešitev

V tem poglavju smo grafično predstavili hitrosti posameznih funkcij. Že mala sprememba v kodi lahko pomeni veliko spremembo v hitrosti v pozitivno ali negativno smer. Novejše funkcije so izpeljanke iz prejšnjih starejših funkcij. Te sorodne funkcije imajo tudi enako obliko grafov. Obliko grafov smo razložili v besedilu.

Pri vseh grafih se moramo zavedati, da so meritve opravljene le za določene vrednosti vzdolž x osi. Da bi se videl globalen trend, smo te točke povezali in meritve prikazali kot zvezne funkcije. Kjer je potrebno, smo namesto tega pristopa uporabili stolpične diagrame.

4.1 Analiza izbranih rešitev

Kot vemo iz prejšnjih poglavij, je funkcija GS0 namenjena preverjanju rešitev. Funkciji GS1 in GS2 sta prvi funkciji napisani z OpenMP anotacijami. Pri pisanju teh dveh funkcij smo se še spoznavali z OpenMP knjižnico. Funkciji GS8 in GS9 pa sta naši najhitrejši realizaciji.

V tem poglavju smo analizirali zgoraj omenjene funkcije. Za vsako izmed naštetih funkcij imamo izmerjen čas, ki ga porabi za izračun Gram-Schmidtovega postopka. Za vsako funkcijo smo izpeljali formulo, ki prešteje število operacij v plavajoči vejici. Za funkcije GS0, GS1, GS2, GS8 in GS9

smo programsko prešteli dejansko število operacij v plavajoči vejici.

Kot bomo videli na spodnjih primerih, nam grafi ujemanja modela časovne funkcije z izmerjenimi vrednostmi povedo, da je naša formula, čez palec, dobra ocena za časovno zahtevnost naših funkcij, ne pa vedno tudi za oceno hitrosti programa, a o tem kasneje.

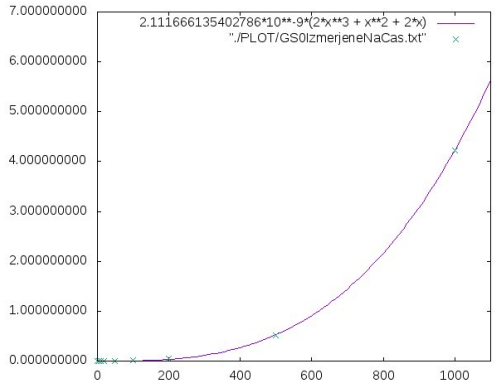
4.1.1 Izračun operacij v plavajoči vejici za funkcijo GS0

Časovno zahtevnost funkcije GS0 ocenimo s številom operacij v plavajoči vejici (FLOP), glede na število in dolžino vektorja, torej n. Celoten izračun števila operacij, kjer $Z_m(n)$ označuje število operacij v zanki z oznako $ZANKA_m$ v izvorni kodi funkcije GS0 (Priloga A.1), je sledeč:

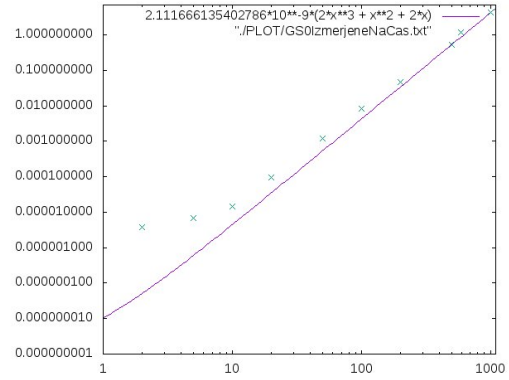
$$
\begin{aligned}
Z_1(n) &= 2n \\
Z_2(n) &= 2 + n \\
Z_{3.1.1}(n) &= 2n \\
Z_{3.1.2}(n) &= 2n \\
Z_{3.1}(n,i) &= i(Z_{3.1.1}(n) + Z_{3.1.2}(n)) = i(2n + 2n) = 4ni \\
Z_{3.2}(n) &= 2n \\
Z_{3.3}(n) &= 2 + n \\
Z_3(n) &= \sum_{i=1}^{n-1}(Z_{3.1}(n,i) + Z_{3.2}(n) + Z_{3.3}(n)) \\
&= (n-1)(Z_{3.2}(n) + Z_{3.3}(n)) + \sum_{i=1}^{n-1} Z_{3.1}(n,i) \\
&= (n-1)(3n+2) + \sum_{i=1}^{n-1} 4ni \\
&= 3n^2 - n - 2 + 4n\frac{n(n-1)}{2} \\
&= 3n^2 - n - 2 + 2n^3 - 2n^2 \\
&= 2n^3 + n^2 - n - 2
\end{aligned}
$$

$$\begin{aligned} GS0(n) &= Z_1(n) + Z_2(n) + Z_3(n) \\ &= 2n + (2+n) + 2n^3 + n^2 - n - 2 \\ &= 2n^3 + n^2 + 2n \end{aligned}$$

Funkcija GS0 zahteva $2n^3 + n^2 + 2n$ FLOPs.



Slika 4.1: Ujemanje modela in meritve za funkcijo GS0.



Slika 4.2: Ujemanje modela in meritve za funkcije GS0.

Na Sliki 4.1 je prikazano ujemanje izmerjenega časa s časom predvidenim po modelu, ki predstavlja število FLOP-ov potrebnih za izračun Gram-Schmidtovega postopka. Slika 4.2 prikazuje isto na logaritemski skali.

4.1.2 Izračun operacij v plavajoči vejici za funkcijo GS1

Spodaj je izračun števila operacij s plavajočo vejico za funkcijo GS1, kjer p predstavlja število niti. Koda funkcije GS1 se nahaja v Dodatku A.2.

$$\begin{aligned} Z_1(n) &= 2n \\ Z_2(n) &= 2 + n/p \\ Z_{3.1.1}(n) &= 2n \end{aligned}$$

$$
\begin{aligned}
Z_{3.1.2}(n) &= 2n \\
Z_{3.1}(n,i) &= i(Z_{3.1.1}(n) + Z_{3.1.2}(n)) = i(2n + 2n) = 4ni \\
Z_{3.2}(n) &= 2n \\
Z_{3.3}(n) &= 2 + n \\
Z_3(n) &= \sum_{i=1}^{n-1}(Z_{3.1}(n,i) + Z_{3.2}(n) + Z_{3.3}(n)) \\
&= (n-1)(Z_{3.2}(n) + Z_{3.3}(n)) + \sum_{i=1}^{n-1} Z_{3.1}(n,i) \\
&= (n-1)(2n + 2 + n) + \sum_{i=1}^{n-1} 4ni \\
&= (n-1)(3n+2) + 4n\frac{n(n-1)}{2} \\
&= 3n^2 - n - 2 + 2n^3 - 2n^2 \\
&= 2n^3 + n^2 - n - 2
\end{aligned}
$$

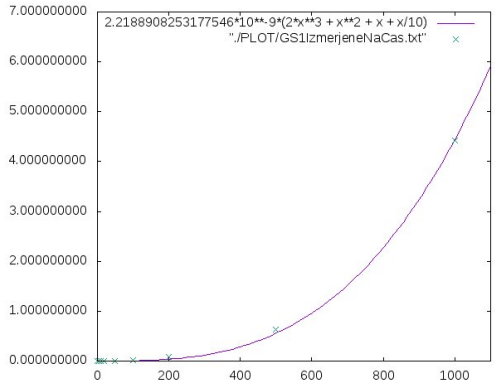
$$
\begin{aligned}
GS1(n) &= Z_1(n) + Z_2(n) + Z_3(n) \\
&= 2n + 2 + n/p + 2n^3 + n^2 - n - 2 \\
&= 2n^3 + n^2 + n + n/p
\end{aligned}
$$

Ujemanje izmerjenih časov s predvidenim je predstavljeno na Sliki 4.3. Ujemanje časov z logaritemsko skalo je na Sliki 4.4.

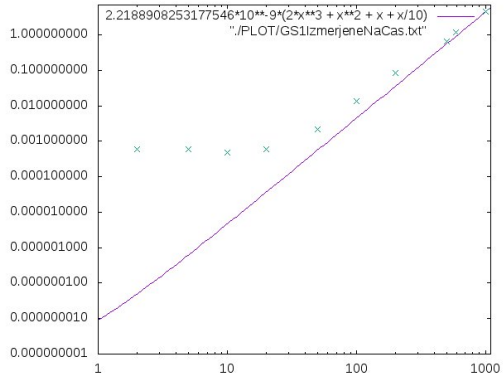
4.1.3 Izračun operacij v plavajoči vejici za funkcijo GS2

Izračun operacij s plavajočo vejico za funkcijo GS2.

$$
\begin{aligned}
Z_1(n) &= 2n/p \\
Z_2(n) &= 2 + n/p \\
Z_{3.1.1}(n) &= 2n/p
\end{aligned}
$$



Slika 4.3: Ujemanje modela in meritve za funkcijo GS1.

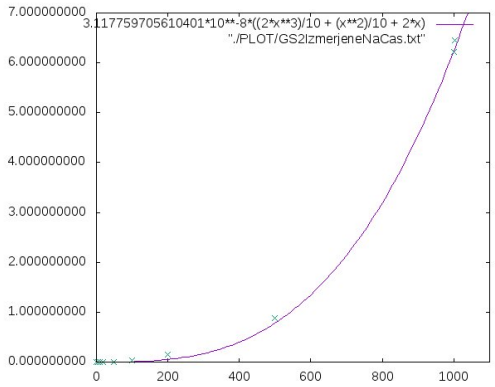


Slika 4.4: Ujemanje modela in meritve za funkcijo GS1.

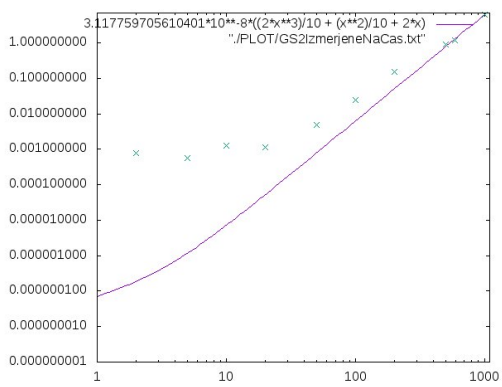
$$
\begin{aligned}
Z_{3.1.2}(n) &= 2n/p \\
Z_{3.1}(n,i) &= i(Z_{3.1.1}(n) + Z_{3.1.2}(n)) = i(2n/p + 2n/p) = 4ni/p \\
Z_{3.2}(n) &= 2n/p \\
Z_{3.3}(n) &= 2 + n/p \\
Z_3(n) &= \sum_{i=1}^{n-1}(Z_{3.1}(n,i) + Z_{3.2}(n) + Z_{3.3}(n)) \\
&= (n-1)(Z_{3.2}(n) + Z_{3.3}(n)) + \sum_{i=1}^{n-1} Z_{3.1}(n,i) \\
&= (n-1)(2n/p + 2 + n/p) + \sum_{i=1}^{n-1} 4ni/p \\
&= (n-1)(3n/p + 2) + 4n/p\frac{n(n-1)}{2} \\
&= 3n^2/p + 2n - 3n/p - 2 + 2n^3/p - 2n^2/p \\
&= 2n^3/p + n^2/p + 2n - 3n/p - 2
\end{aligned}
$$

$$
\begin{aligned}
GS2(n) &= Z_1(n) + Z_2(n) + Z_3(n) \\
&= 2n/p + 2 + n/p + 2n^3/p + n^2/p + 2n - 3n/p - 2 \\
&= 2n^3/p + n^2/p + 2n
\end{aligned}
$$

Ujemanje izmerjenih časov s predvidenim je predstavljeno na Slikah 4.5 in 4.6.



Slika 4.5: Ujemanje modela in meritve za funkcijo GS2.



Slika 4.6: Ujemanje modela in meritve za funkcijo GS2.

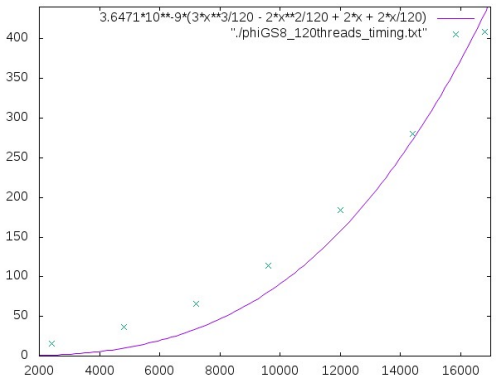
4.1.4 Izračun operacij v plavajoči vejici za funkcijo GS8

Predpostavimo, da je n/p vedno celo število, kjer n predstavlja število vektorjev in p število niti. Sledi izračun števila operacij s plavajočo vejico za funkcijo GS8.

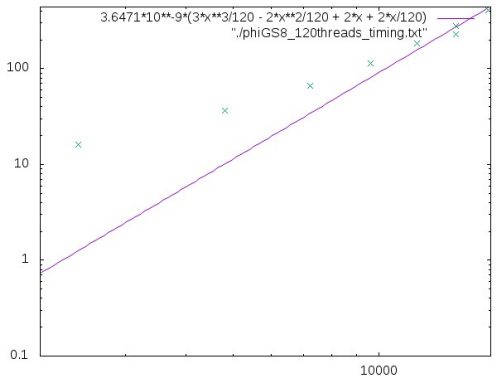
$$
\begin{aligned}
Z_1(n) &= 2n/p \\
Z_2(n) &= 2 + n/p \\
Z_{3.1.1}(n) &= 2n \\
Z_{3.1.2}(n) &= 4n \\
Z_{3.1}(n,i) &= i(Z_{3.1.1}(n) + Z_{3.1.2}(n))/p = i(2n + 4n)/p = 6ni/p \\
Z_{3.2}(n) &= 2 + n/p \\
Z_3(n) &= \sum_{i=1}^{n-1}(Z_{3.1}(n,i) + Z_{3.2}(n))
\end{aligned}
$$

$$
\begin{aligned}
&= (n-1)Z_{3.2}(n) + \sum_{i=1}^{n-1} Z_{3.1}(n,i) \\
&= (n-1)(2+n/p) + \sum_{i=1}^{n-1} 6ni/p \\
&= 2n - 2 + n^2/p - n/p + 6n/p\frac{n(n-1)}{2} \\
&= 2n - 2 + n^2/p - n/p + 3n^3/p - 3n^2/p \\
&= 3n^3/p - 2n^2/p + 2n - n/p - 2
\end{aligned}
$$

$$
\begin{aligned}
GS8(n) &= Z_1(n) + Z_2(n) + Z_3(n) \\
&= 2n/p + 2 + n/p + 3n^3/p - 2n^2/p + 2n - n/p - 2 \\
&= 3n^3/p - 2n^2/p + 2n + 2n/p
\end{aligned}
$$



Slika 4.7: Ujemanje modela in meritve za funkcijo GS8.



Slika 4.8: Ujemanje modela in meritve za funkcijo GS8.

Ujemanje izmerjenih časov s predvidenim je predstavljeno na Slikah 4.7 in 4.8. Zelene točke predstavljajo izmerjen čas, vijolična funkcija pa predstavlja našo oceno za število operacij s plavajočo vejico. Razlog za rahlo odstopanje točk od funkcije je v tem, da ocena zajema samo število operacij s plavajočo vejico, ne zajema pa sinhronizacij niti, povečevanj števcev, shranjevanj podatkov in primerjave števil.

4.1.5 Izračun operacij v plavajoči vejici za funkcijo GS9

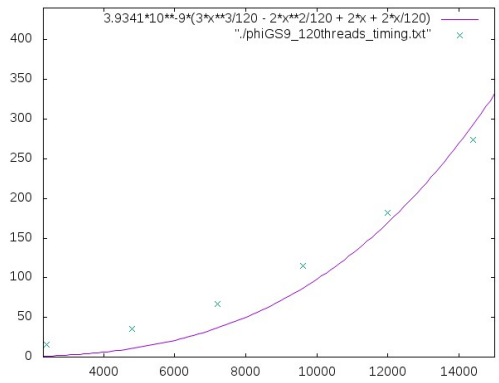
Predpostavimo, da je n/p vedno celo število, kjer n predstavlja število vektorjev in p število niti. Sledi izračun števila operacij s plavajočo vejico, ki je enak kot izračun za funkcijo GS8. Sprememba je samo v prvem in zadnjem normiranju, kjer se le-to v celoti izvede v `for` zanki z OpenMP anotacijo za vzporednost.

$$
\begin{aligned}
Z_1(n) &= 2n/p \\
Z_2(n) &= 2 + n/p \\
Z_{3.1.1}(n) &= 2n \\
Z_{3.1.2}(n) &= 4n \\
Z_{3.1}(n,i) &= i(Z_{3.1.1}(n) + Z_{3.1.2}(n))/p = i(2n+4n)/p = 6ni/p \\
Z_{3.2}(n) &= 2 + n/p \\
Z_3(n) &= \sum_{i=1}^{n-1}(Z_{3.1}(n,i) + Z_{3.2}(n)) \\
&= (n-1)Z_{3.2}(n) + \sum_{i=1}^{n-1} Z_{3.1}(n,i) \\
&= (n-1)(2+n/p) + \sum_{i=1}^{n-1} 6ni/p \\
&= 2n - 2 + n^2/p - n/p + 6n/p\frac{n(n-1)}{2} \\
&= 2n - 2 + n^2/p - n/p + 3n^3/p - 3n^2/p \\
&= 3n^3/p - 2n^2/p + 2n - n/p - 2
\end{aligned}
$$

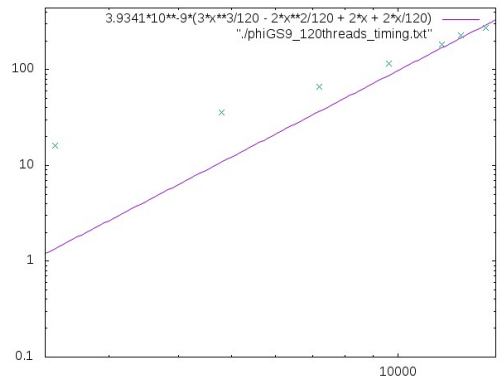
$$
\begin{aligned}
GS9(n) &= Z_1(n) + Z_2(n) + Z_3(n) \\
&= 2n/p + 2 + n/p + 3n^3/p - 2n^2/p + 2n - n/p - 2 \\
&= 3n^3/p - 2n^2/p + 2n + 2n/p
\end{aligned}
$$

Ujemanje izmerjenih časov s predvidenim je predstavljeno na Slikah 4.9

in 4.10. Zelene točke predstavljajo izmerjen čas, vijolična funkcija pa predstavlja našo oceno za število operacij s plavajočo vejico.



Slika 4.9: Ujemanje modela in meritve za funkcijo GS9.



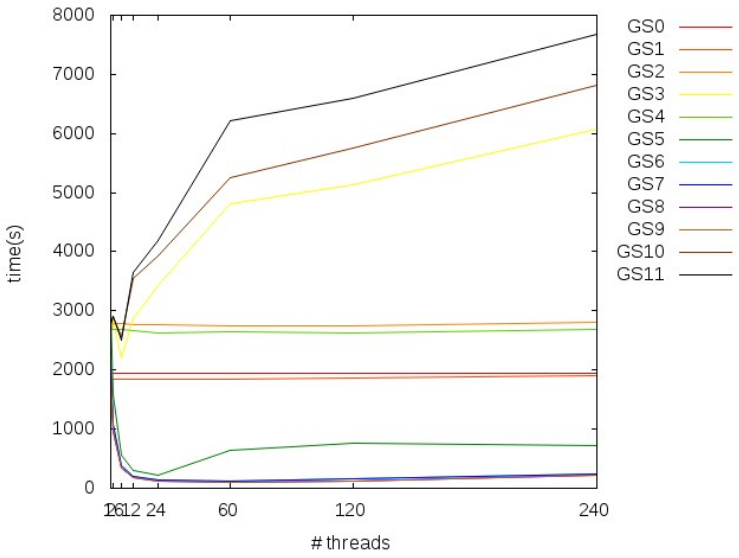
Slika 4.10: Ujemanje modela in meritve za funkcijo GS9.

4.2 Primerjava časa izvajanja med posameznimi rešitvami

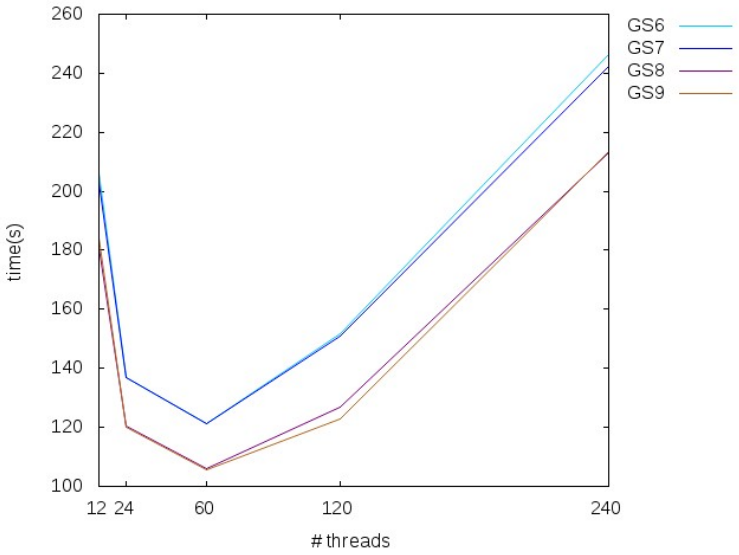
Za lažjo predstavo, kako hitri so posamezni algoritmi, si poglejmo graf časa izvajanj vseh funkcij za velikost vektorjev 10000 na Intel Xeon Phi koprocesorju. Ti grafi so prikazani na Slikah od 4.11 do 4.14.

Kot se vidi na Sliki 4.11, so funkcije GS2, GS3, GS4, GS10 in GS11 mnogo počasnejše tudi od sekvenčne funkcije GS0. Kljub temu, da GS2 računa vse skalarne produkte in projekcije vzporedno, je še vedno mnogo počasnejša od funkcije GS0. Razlog je v izvrševanju projekcij na neortogonalne vektorje. V funkciji GS2 se vsak ortogonalen vektor zaporedno projecira na vse ostale vektorje. To smo dosegli z openMP `ordered` anotacijo. Anotacija `ordered` se deklarira ob anotaciji `parallel` nato pa se znotraj `parallel` bloka deklarira `ordered` blok. Ta blok se izvede kot, da je napisan zaporedno.

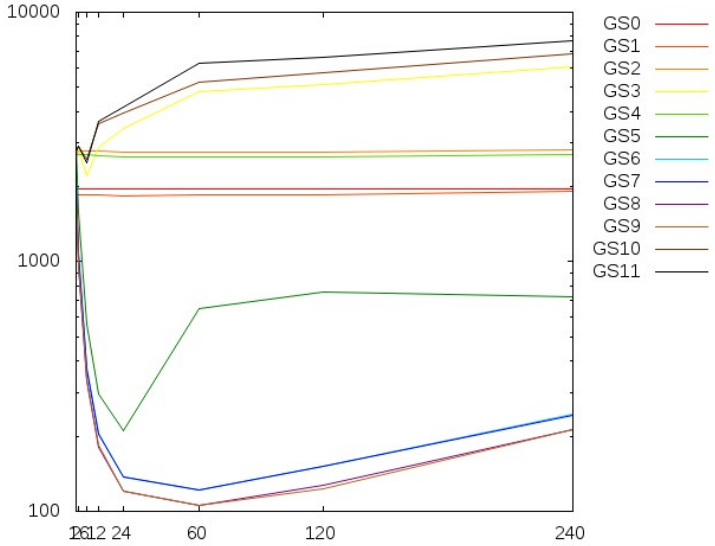
Funkcija GS4 ima enako napako kot funkcija GS2. Razlika med njima je ta, da funkcija GS2 uporablja OpenMP `ordered` anotacijo za zaporedno



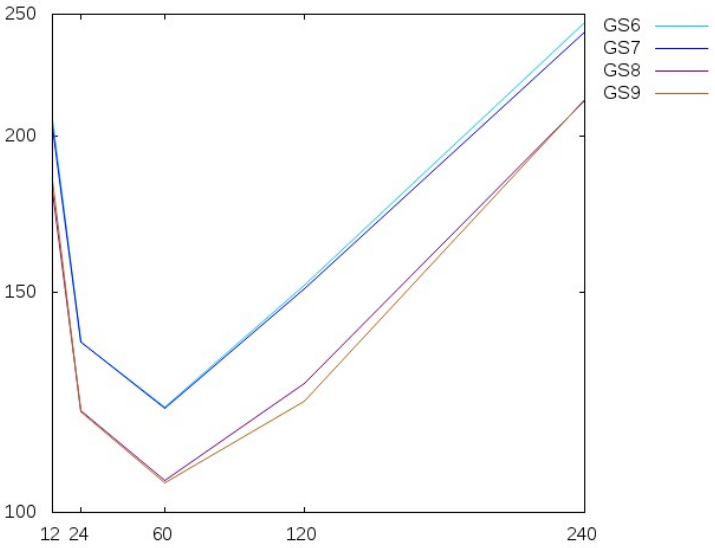
Slika 4.11: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 10000 x 10000.



Slika 4.12: Graf časa izvajanj funkcij od GS6 do GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.



Slika 4.13: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 10000 x 10000.



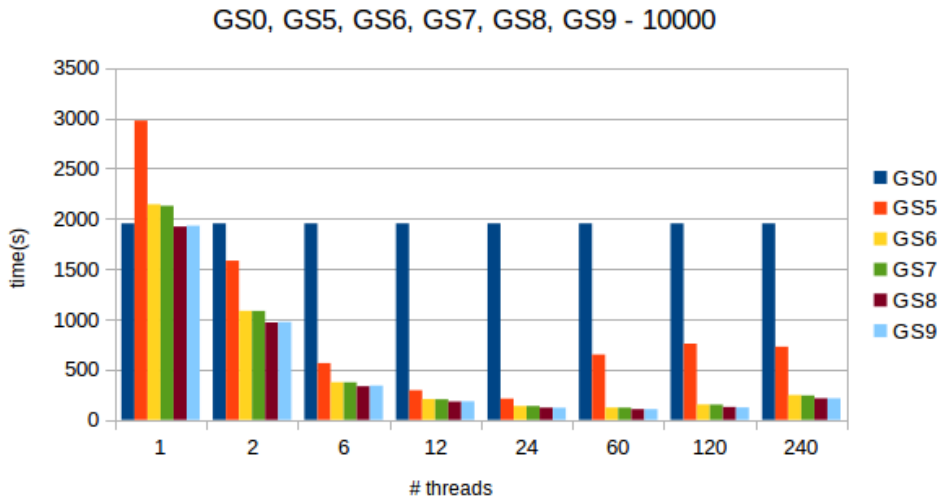
Slika 4.14: Graf časa izvajanj funkcij GS6 – GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.

izvajanje projekcij, medtem ko funkcija GS4 projekcije izvaja vzporedno. Se pravi funkcija GS4 vzporedno projecira ortogonalne vektorje na trenuten vektor. Ker pa zunanja zanka funkcije GS4 porabi vse niti, ki so na voljo, se tudi projekcije ne morejo izvršiti vzporedno, ker jim primanjkuje virov.

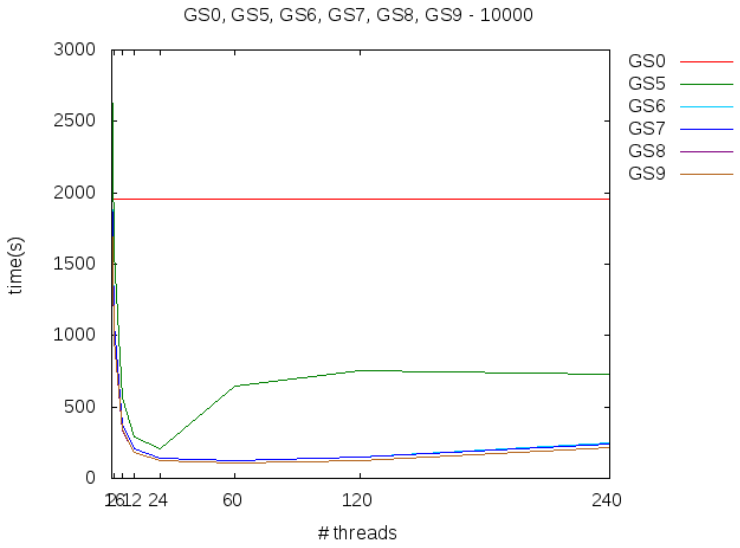
Kot smo ugotovili že iz zgornjih 2 funkcij, izvajanje projekcij zaporedno ni hitra rešitev. Enak problem imajo tudi funkcije GS3, GS10 in GS11, kjer se projekcije ortogonalnih vektorjev izvršijo zaporedno brez OpenMP anotacij. Iz Slike 4.11 se lepo vidi, da funkcije GS3, GS10 in GS11 spadajo v skupino brez zaporednih OpenMP anotacij, medtem ko funkciji GS2 in GS4 uporabljata zaporedno OpenMP anotacijo pri izvajanju projekcij. Iz grafa lahko sklepamo, da je uporaba OpenMP `ordered` anotacije, še vedno hitrejša kot uporaba funkcije brez te anotacije. Anotacija `ordered` še vedno omogoča izvajanje kode znotraj `parallel` bloka vzporedno do neke mere. Knjižnica OpenMP poskrbi, da vedno obstaja nit, ki izvaja sekcijo `ordered`, ki se mora izvesti kot naslednja glede na ordered blok. Ker pa so vse zgoraj naštete funkcije slabše od zaporedne implementacije GS0, jih v nadaljevanju ne bomo več omenjali in prikazovali.

Kot smo že zgoraj ugotovili, uporaba OpenMP anotacij ni vedno najboljša. Uporaba anotacij pri eni niti se ne obnese niti za hitrejše funkcije Slike od 4.15 do 4.17. Ampak to nas ne zanima toliko, saj smo izbrali knjižnico OpenMP z namenom, da pohitrimo Gram-Schmidtov postopek na več jedrih/nitih.

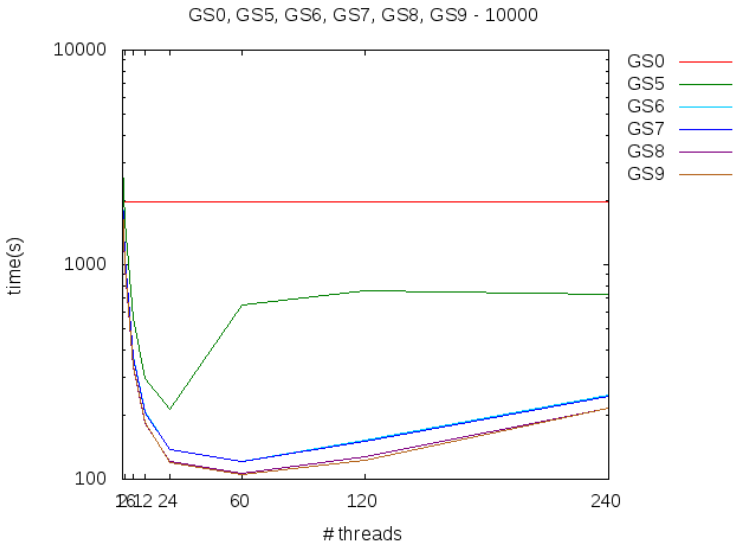
Na Sliki 4.15 se opazi, da je funkcija GS5 od 24 niti naprej veliko slabša kot preostale funkcije. Razlika med funkcijo GS5 in ostalimi funkcijami je, da funkcija GS5 izvede vse projekcije ortogonalnih vektorjev znotraj enega stavka `parallel for`. Funkcija GS5 pa znotraj tega stavka `parallel for` ponovno ustvari nove vzporedne sekcije in zahteva nove niti. Kot se vidi na Grafu 4.15, to ni problem, do 24 niti, saj tu zunanja zanka ustvari 24 niti, nato pa vsaka notranja zanka ustvari novih 24 niti, kar skupaj nanese 576 niti. Intel Xeon Phi pa ima skupna na razpolago maksimalno 240 niti. Pri tem številu Intel Xeon Phi še nekako lahko izvaja preklapljanje med konteksti,



Slika 4.15: Graf časa izvajanj funkcij GS5 do GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.



Slika 4.16: Graf časa izvajanj funkcij GS5 do GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.



Slika 4.17: Graf časa izvajanj funkcij GS5 do GS9 na Intel Xeon Phi za velikost matrike 10000 x 10000.

brez da bi se to poznalo na času izračuna. Če pa vzamemo 60 niti, to nanese 3600 niti, znotraj 2 vzporednih blokov, kar pa je že preveč in preklapljanje med konteksti povzroča veliko izgubo časa. Ostale funkcije (GS6 – GS9) pa si najprej pravično razdelijo delo, tako da vsaka opravi svojih 416 projekcij. Zatem pa vedno projecirajo samo en vektor naenkrat, ampak znotraj te zanke si razdelijo delo za izračun skalarnega produkta in projekcije. Pri velikosti vektorja 10000 se to že zelo pozna.

Za konec bomo primerjali še graf funkcij od GS6 do GS9. Glej Sliki 4.12 in 4.14.

Na le-teh slikah se opazi, da sodita funkciji GS6 in GS7 v eno množico in funkciji GS8 in GS9 v drugo. Glavna razlika med temi funkcijami je v tem, da sta GS6 in GS7 klasična Gram-Schmidt postopka, medtem ko sta funkciji GS8 in GS9 modificirana Gram-Schmidt postopka.

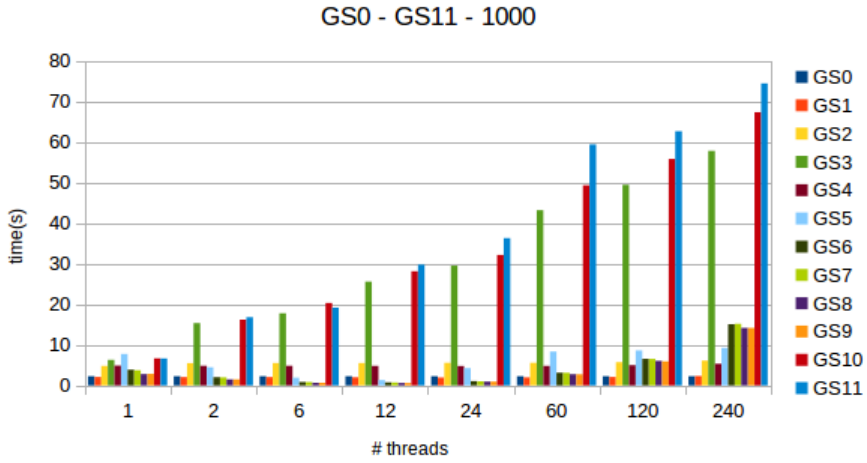
Kot smo že napisali v Poglavju 3.3, sta funkciji GS6 in GS7 enaki. Razlikujeta se samo v definiciji vhodne in izhodne matrike. Matriki v funkciji GS7 smo dodali `__attribute__((aligned(64)))`. Kot vidimo iz Slike 4.12,

pa to nima vpliva na samo hitrost algoritma. Kot kaže je prevajalnik že brez naše anotacije poravnal matriki.

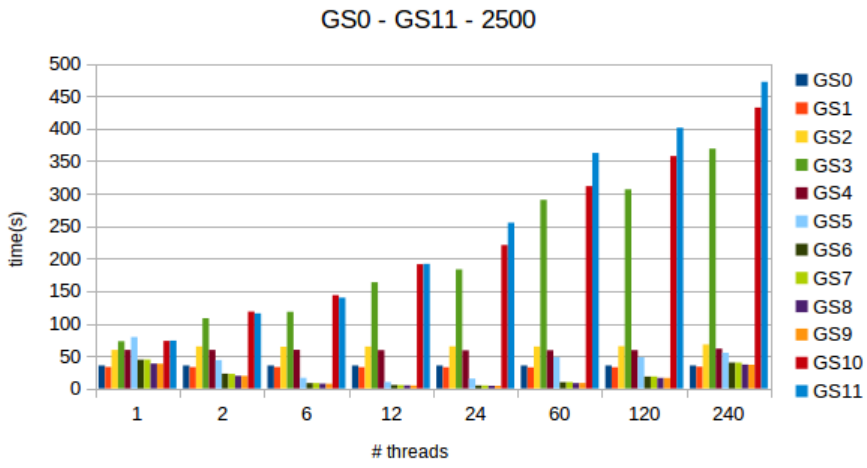
Funkcija GS9 je izpeljanka funkcije GS8, zato skoraj identičen graf ni nobeno presenečenje. V GS9 smo združili skalarni produkt nazaj pod eno for zanko z eno OpenMP *parallel* anotacijo. Enako smo naredili za vsa normiranja. Kot že omenjeno, sta funkciji GS8 in GS9 predstavnici modificiranega Gram-Schmidtovega postopka. Ker se računanje opravlja nad velikimi vektorji, ne preseneča, da ima lokalni dostop do pomnilnika kar veliko vlogo pri hitrosti izračuna. Razlika v času med funkcijama GS6 in GS8 je v povprečju 13 % v prid funkciji GS8. Največja razlika se naredi pri računanju skalarnega produkta, kjer za podatke ne rabimo več dveh tabel, ampak sedaj beremo podatke iz ene tabele.

4.3 Primerjava meritev za različne velikosti problema

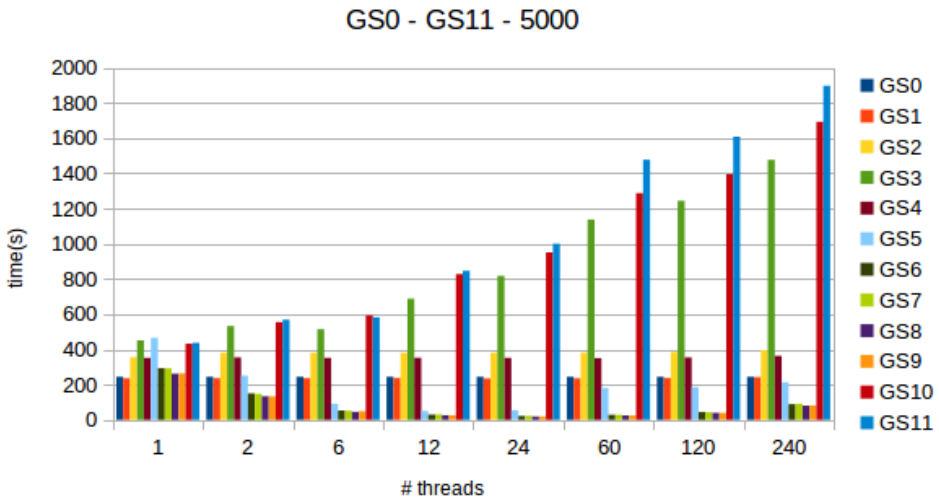
Na začetku smo testirali naše GS funkcije na manjših vektorjih in jih nato postopno povečevali. Začeli smo z manjšimi matrikami zato, da smo lažje in hitreje preverili pravilnost izračuna. Vmesni koraki med velikostjo 1000 vektorjev in 10000 vektorjev pa nam služijo predvsem za oris obnašanja algoritmov, ko se vhodni podatki povečujejo. Kot lahko vidimo iz Slik 4.18 do 4.22 imajo vse slike približno enako obliko. Slike od 4.23 do 4.27 prikazujejo iste podatke na zvezni x osi in logaritemski skali y osi.



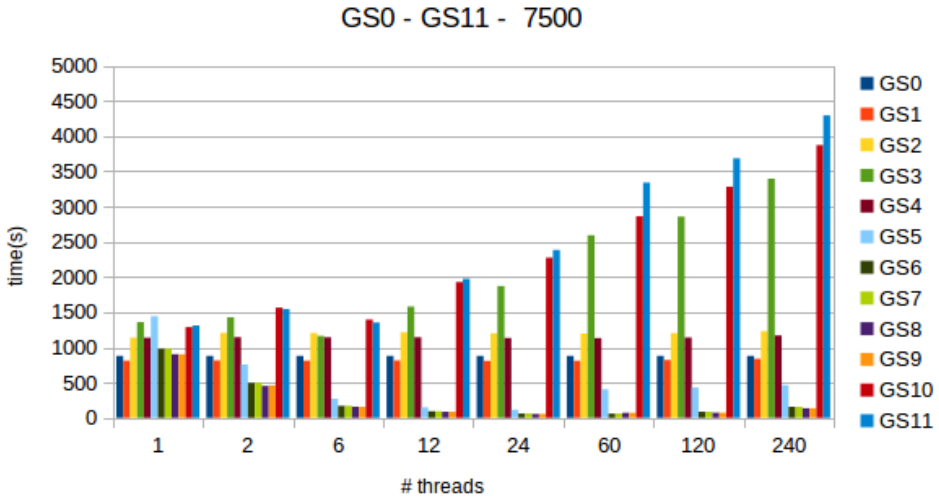
Slika 4.18: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 1000 x 1000.



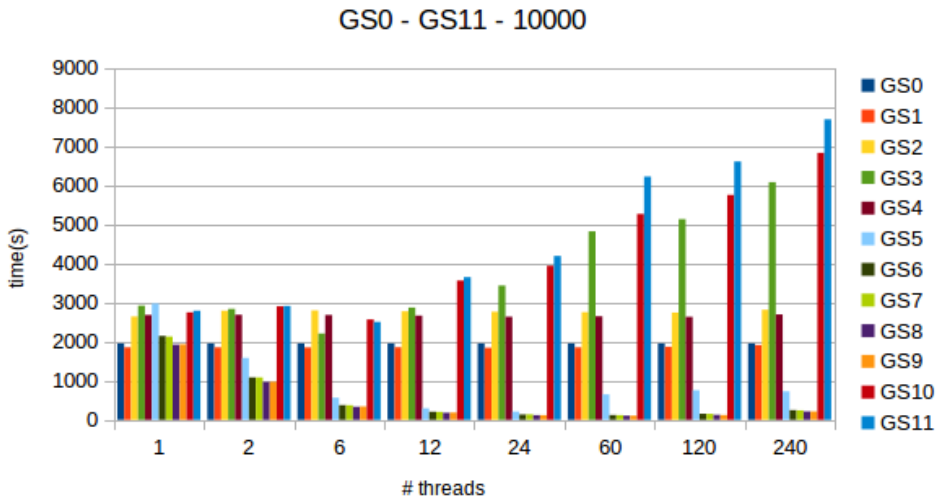
Slika 4.19: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 2500 x 2500.



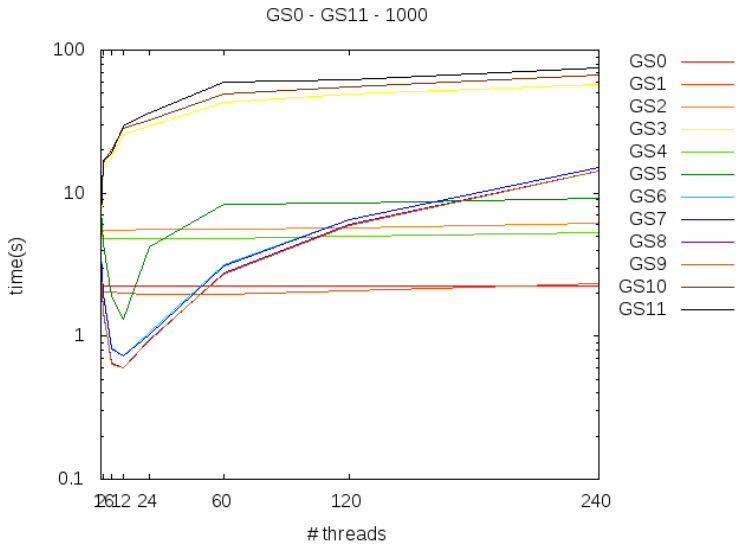
Slika 4.20: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 5000 x 5000.



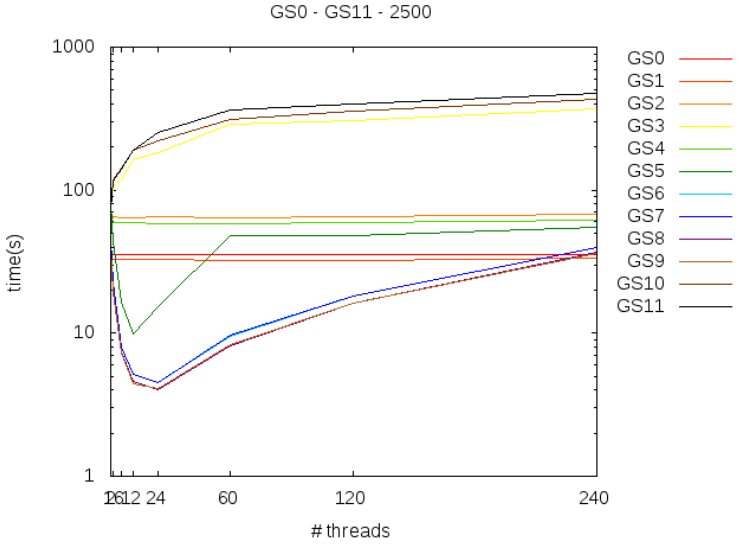
Slika 4.21: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 7500 x 7500.



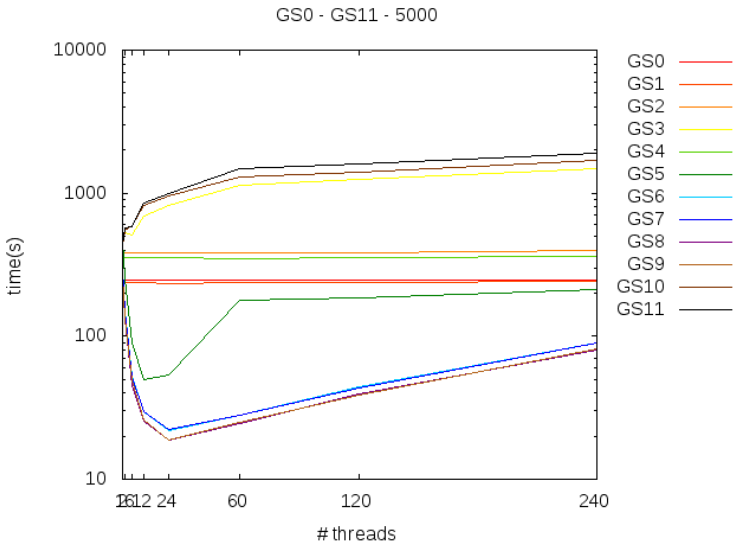
Slika 4.22: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 10000 x 10000.



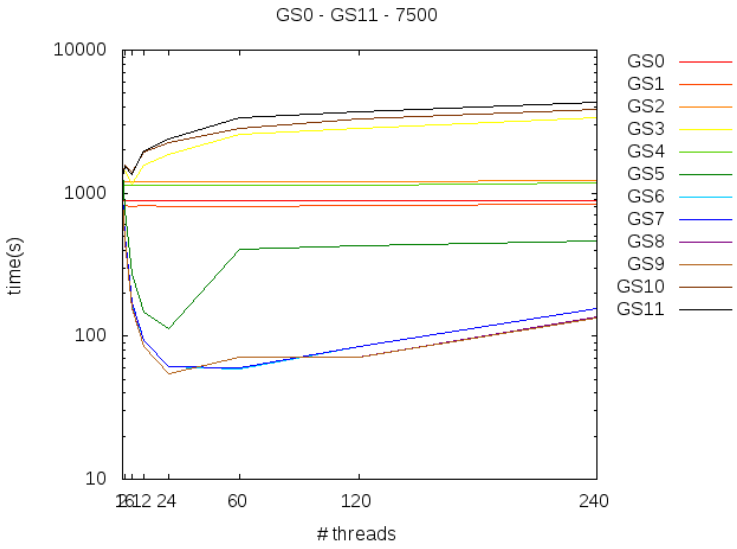
Slika 4.23: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 1000 x 1000.



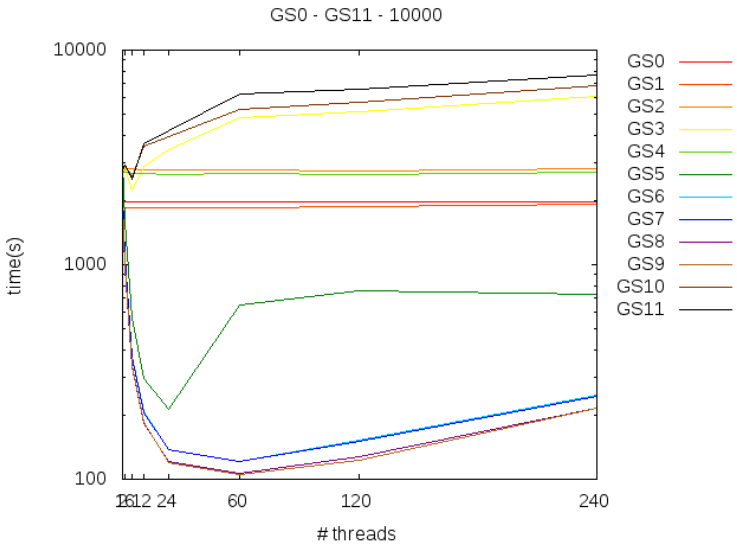
Slika 4.24: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 2500 x 2500.



Slika 4.25: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 5000 x 5000.



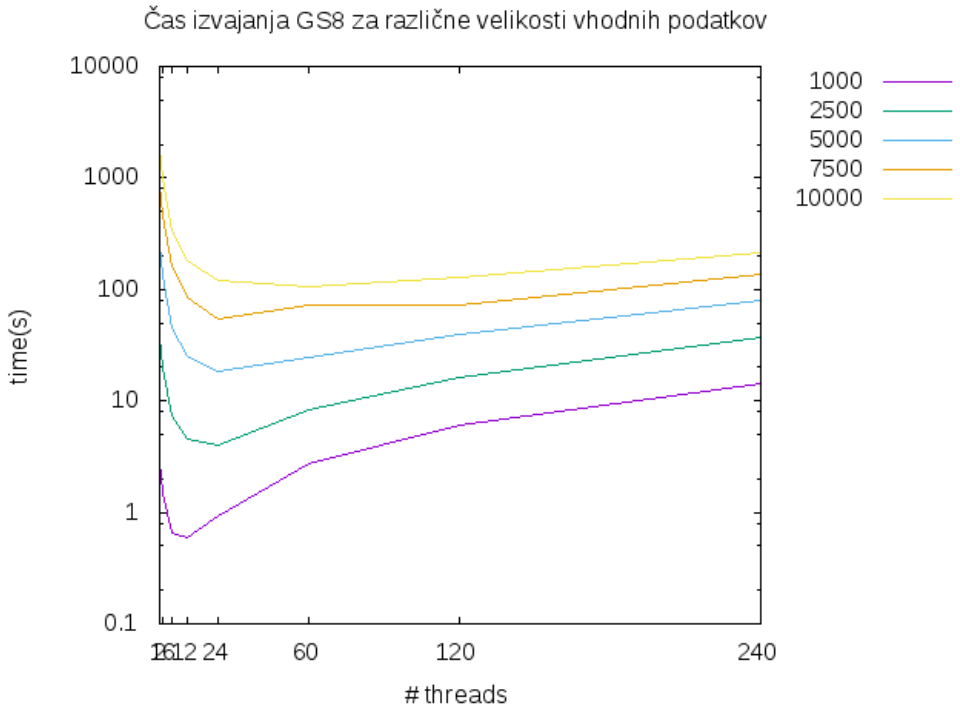
Slika 4.26: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 7500 x 7500.



Slika 4.27: Graf časa izvajanj vseh funkcij na Intel Xeon Phi za velikost matrike 10000 x 10000.

Pri računanju z eno nitjo za vse različne velikosti vhodni podatkov so grafi precej izenačeni. Pri večanju števila niti se hitro pokažejo slabosti funkcij GS3, GS10 in GS11. Pri manjših vhodnih podatkih in nizkem številu še ni opaziti odstopanja funkcij GS1, GS2, GS4, ki pa se z večanjem vektorjev hitro razkrijejo.

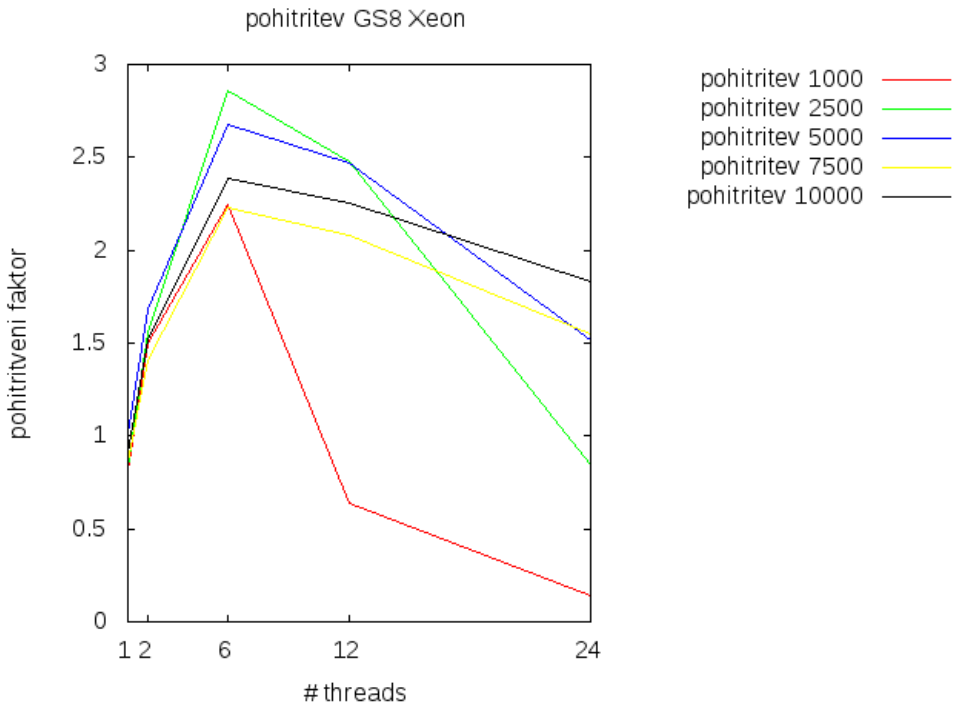
Slika 4.28 prikazuje čas izvajanja funkcije GS8 na Intel Xeon Phi koprocesorju za vse velikosti vhodnih matrik. Velikost vhodnih podatkov je premo sorazmerna s potrebnim številom niti za najhitrejše izvajanje. Za majhno število vhodnih vektorjev potrebujemo manjše število niti, da dosežemo maksimalno zmogljivost. Tako se GS8 funkcija najhitreje izvede pri 12 nitih za 1000 vhodnih vektorjev, pri 24 nitih za 2500, 5000 in 7500 vektorjev in pri 60 nitih za 10000 vhodnih vektorjev.



Slika 4.28: Graf časa izvajanj najhitrejše funkcije GS8 na Intel Xeon Phi za vse velikosti matrik.

4.4 Intel Xeon ali Intel Xeon Phi?

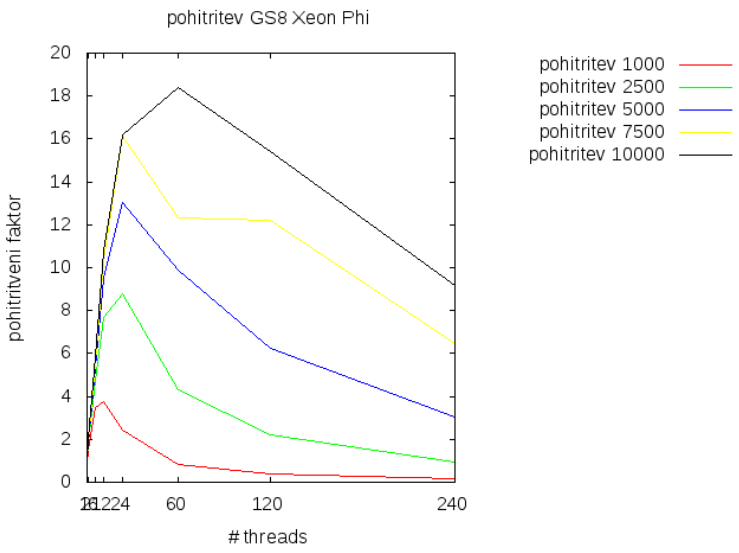
V tem razdelku smo utemeljili izbiro med Intel Xeon in Intel Xeon Phi koprocesorjem. Odločitev je temeljila na faktorju pohitritve, učinkovitosti in ceni. V absolutnem pogledu pa je najbolje zagnati funkcijo GS8 na Intel Xeon Phi pri 60 nitih, saj le-ta kombinacija izvede Gram-Schmidtov algoritem najhitreje. Najhitrejša funkcija na Xeonu izvede Gram-Schmidtov postopek na 10000 vektorjih v 312 sekundah, medtem ko je najhitrejša funkcija na Xeon Phiju izvede Gram-Schmidtov postopek na 10000 vektorjih v 106 sekundah. Intel Xeon Phi nalogo opravi v trikrat krajšem času kot Intel Xeon procesor.



Slika 4.29: Graf pohitritev GS8 funkcije glede na GS0 funkcijo na Xeon za različne vhodne podatke.

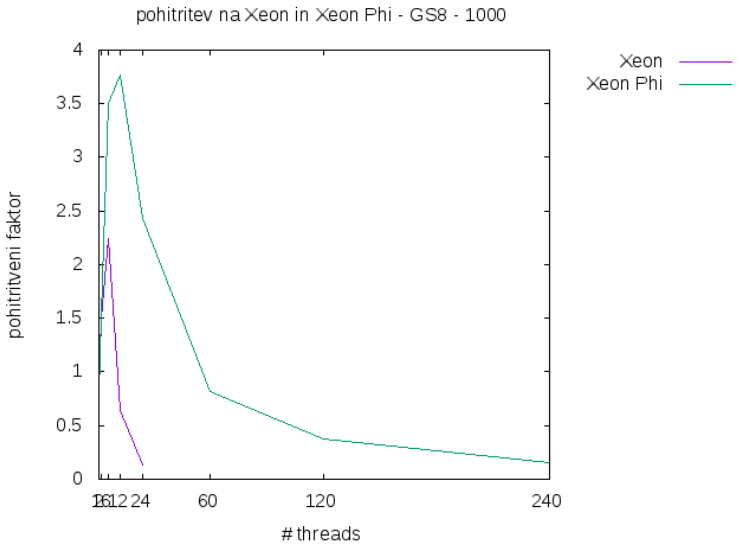
4.4.1 Pohitritev

Sedaj vemo, da je v absolutnem smislu najbolje zagnati Gram-Schmidtov postopek na Intel Xeon Phi pri 60 nitih. Ne vemo pa, ali ima Xeon Phi tudi največjo pohitritev. Na Sliki 4.29 vidimo, da je imel Xeon največjo pohitritev za velikost problema 2500 z uporabo 6 niti. Intel Xeon Phi pa je največ pridobil na 10000 vektorjih pri 60 nitih. Glej Sliko 4.30.

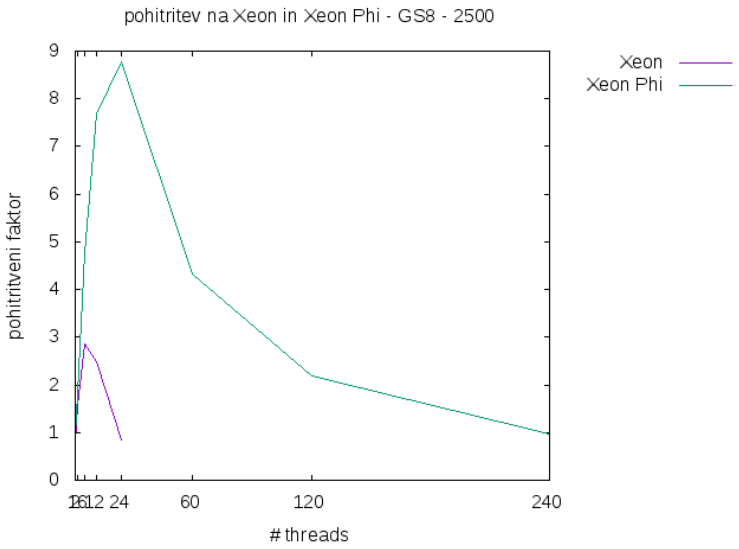


Slika 4.30: Graf pohitritev GS8 funkcije glede na GS0 funkcijo na Xeon Phi za različne vhodne podatke.

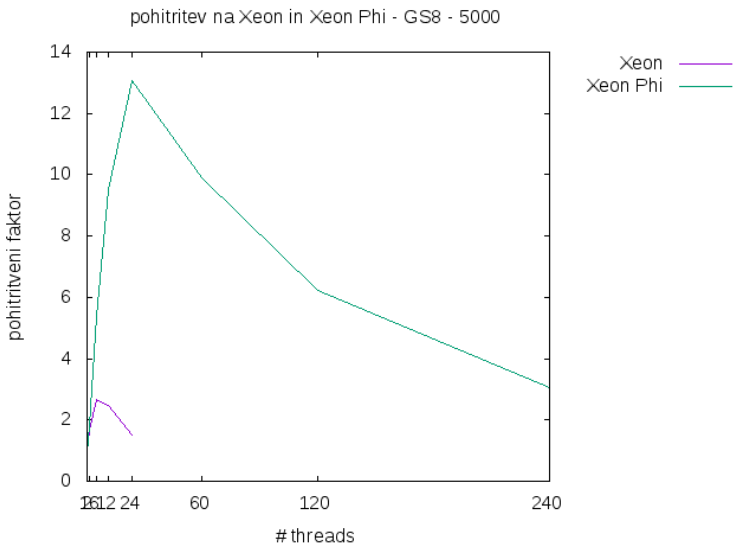
Tudi če pogledamo globalno, ima Intel Xeon Phi boljši faktor pohitritve za vse velikosti vhodnih problemov. Glej Slike od 4.31 do 4.35.



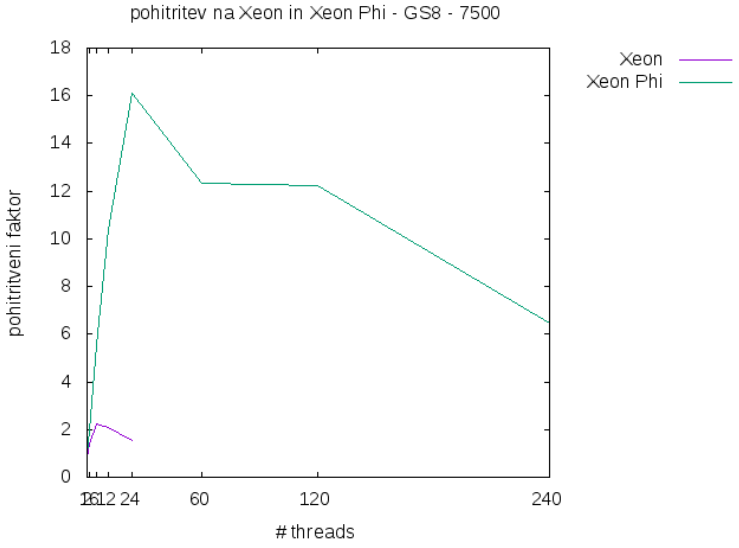
Slika 4.31: Graf pohitritev Xeon proti Xeon Phi za 1000 vhodnih vektorjev.



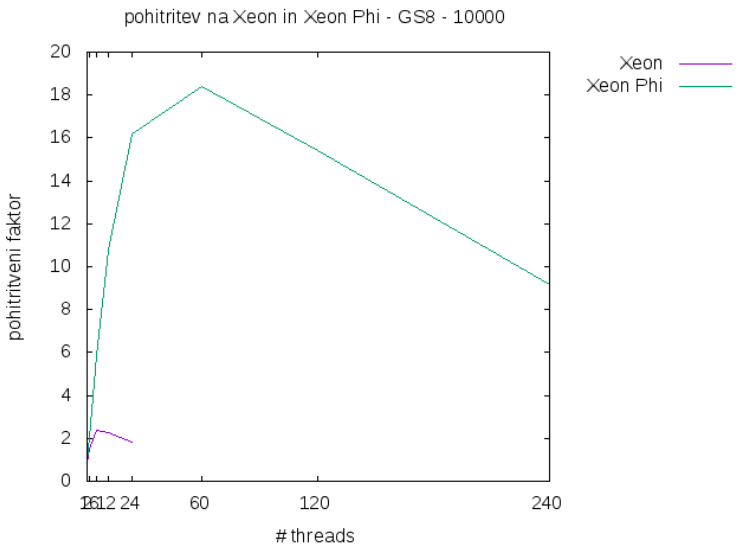
Slika 4.32: Graf pohitritev Xeon proti Xeon Phi za 2500 vhodnih vektorjev.



Slika 4.33: Graf pohitritev Xeon proti Xeon Phi za 5000 vhodnih vektorjev.



Slika 4.34: Graf pohitritev Xeon proti Xeon Phi za 7500 vhodnih vektorjev.



Slika 4.35: Graf pohitritev Xeon proti Xeon Phi za 10000 vhodnih vektorjev.

4.4.2 Učinkovitost

Izračunali smo, da naš algoritem porabi $2n^3 + n^2 + 2n$ FLOPov, kot zahteva osnovni GS (glej Poglavje 4.1.1), za izračun Gram-Schmidtovega postopka, kjer je n enak številu vektorjev. Število FLOP-ov in FLOPS-ov za posamezen n je zapisano v Tabeli 4.1 in 4.2.

Tabela 4.1: Število TFLOP-ov ter TFLOPS-ov glede na n za Intel Xeon.

n	TFLOP	TFLOPS
1000	0,002	0,01019
2500	0,031	0,00845
5000	0,25	0,00652
7500	0,843	0,00644
10000	2	0,00641

Tabela 4.2: Število TFLOP-ov ter TFLOPS-ov glede na n za Intel Xeon Phi.

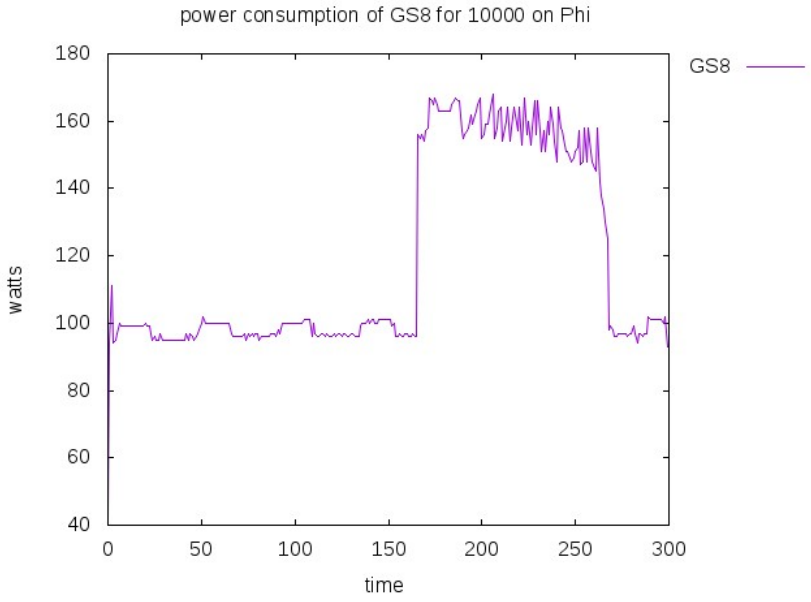
n	TFLOP	TFLOPS
1000	0,002	0,00334
2500	0,031	0,00776
5000	0,25	0,01337
7500	1,843	0,01548
10000	2	0,01885

Teoretična zmogljivost Xeon procesorja je 96GFLOPS [7]. Teoretična zmogljivost Xeon Phi koprocesorja je 1TFLOPS [4].

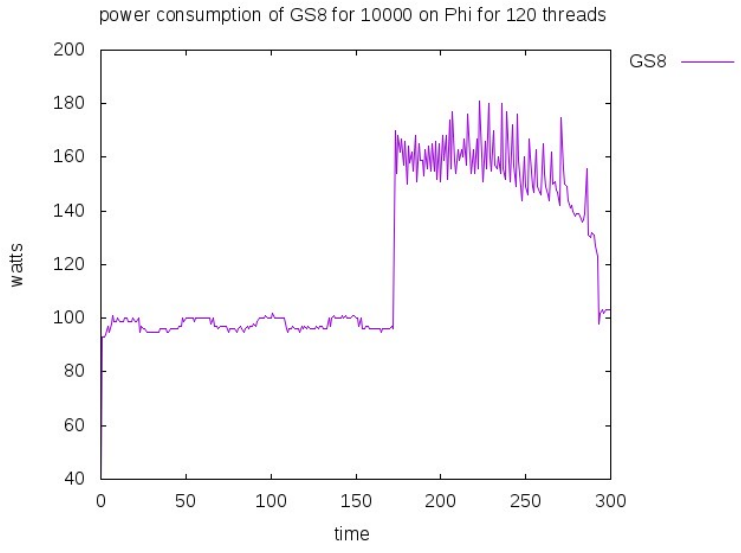
Iz zgornjih tabel je razvidno, da na Intel Xeon maksimalno zmogljivost dosežemo pri manjših velikostih vhodnih vektorjev. Z večanjem števila vhodnih vektorjev pada učinkovitost Intel Xeon procesorja. Pri Intel Xeon Phiju pa dosežemo zgolj 2 % teoretične maksimalne zmogljivosti. Pri Intel Xeon Phiju zmogljivost raste z večanjem števila vhodnih podatkov.

Intel Xeon ima največjo deklarirano porabo 95 W. Intel Xeon Phi pa porabi največ 225 W. Kot vidimo iz Grafa 4.36 Xeon Phi za računanje Gram-Schmidtovega postopka porabi približno 160 W pri 60 nitih. Za 120 niti porabi približno enako, kar vidimo na Sliki 4.37. Za 240 niti se na Sliki 4.38 vidi, da je povprečna poraba še vedno enaka. Iz dolin na grafu se vidi, da koprocesor veliko časa čaka na podatke in takrat zniža porabo.

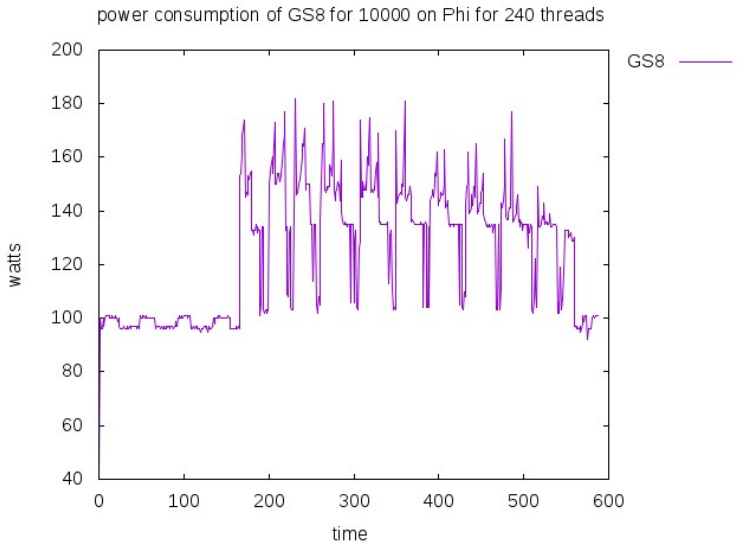
Ker bo Intel Xeon Phi izračunal trikrat hitreje kot Intel Xeon, bi lahko čez palec ocenili, da bo porabil samo 53 W energije, medtem ko bo Intel Xeon porabil vseh 95 W. Intel Xeon Phi je čez palec skoraj dvakrat bolj varčen kot Intel Xeon pri računanju Gram-Schmidtovega postopka.



Slika 4.36: Graf porabe funkcije GS8 na Xeon Phi za 10000 vhodnih vektorjev na 60 nitih.



Slika 4.37: Graf porabe funkcije GS8 na Xeon Phi za 10000 vhodnih vektorjev na 120 nitih.



Slika 4.38: Graf porabe funkcije GS8 na Xeon Phi za 10000 vhodnih vektorjev na 240 nitih.

4.4.3 Cena

Kot smo pokazali, je Intel Xeon Phi hitrejši kot Intel Xeon in porabi za odtenek manj energije. Slaba stran pa je njegova cena. Intel Xeon na katerem smo izvajali meritve, stane 410 $, medtem ko je Intel Xeon Phi kar šestkrat dražji in stane 2649 $.

Poglavje 5

Sklepne ugotovitve

V magistrski nalogi smo implementirali več primerov Gram-Schmidtovega algoritma, med katerimi se je najbolje odrezala funkcija GS8. Glej poglavje 3.4. GS8 je najhitrejša, ker piše in bere iz ene tabele. S tem smo pohitrili dostop do podatkov. Prav tako je najhitrejša, ker pravilno razporedi delo med vse niti. Začetni skalarni produkt in končna normiranja se izvedejo vzporedno za največje možno število elementov v tabeli, preostanek se izvede zaporedno. Prav tako vsaka nit hkrati izvede enako število projekcij ortogonalnih vektorjev. Nato pa vse niti skupaj projecirajo en ortogonalen vektor na trenutni vektor.

Ni presenetljivo, da je funkcija GS8 najhitrejša ravno pri 60 nitih. To je natanko število fizičnih jeder, ki jih ima Intel Xeon Phi. Pri tem številu niti je funkcija najhitrejša, saj privzeto OpenMP nastavi statično razvrstitev (angl. static scheduling). To pa pomeni, da vsaka nit dobi enako količino dela in posledično so vse niti enako obremenjene in porabijo približno enako časa za izračun. Če uporabljamo manj niti, koprocesor ni v celoti izkoriščen, če pa uporabljamo preveč niti, pa prihaja do zastojev pri menjavi podatkov v istem predpomnilniku jedra. Moramo razumeti, da delamo z velikimi tabelami, kjer samo en vektor zasede 80 kB predpomnilnika. Celoten L2 predpomnilnik enega jedra Intel Xeon Phija pa je velikosti 512 kB.

Naša rešitev je primerljiva s prvo rešitvijo iz [10]. Tudi tu se vidi, da

obstaja mejno število jeder oz. niti, kjer večanje števila niti ne pomeni hitrejšega izračuna. Najmanj 6 niti so dodelili enem Intel Xeon X5650 procesorju. Računali so na dveh Xeon X5650 procesorjih. Tako kot pri nas je bila podatkovna širina vodila omejujoči faktor. Bradnes in ostali so v kodo vpeljali računanje po blokih in tako dosegli skoraj idealno pohitritev glede na število niti [10].

Pri razvoju rešitev pa smo naleteli tudi na nekaj problemov. Naleteli smo na tipičen "Segmenatition fault" problem, kjer smo pisali podatke izven tabele. Za majhno tabelo je aplikacija še delovala, pri večjih pa se je začela obnašati zelo nedeterministično.

Naslednji izmed problemov je bil primer, kjer se koda ni več prevedla, če je bila velikost vektorjev večja od 12000.

Pri izvajanju programov in preverjanju izkoristka niti smo ugotovili, da ni nujno, da bo program izkoristil vse niti, ki so mu na voljo. To me je na začetku zelo motilo, dokler si nisem prebral dokumentacije, ki pravi da z nastavitvijo spremenljivke okolja `OMP_NUM_THREADS` poveš maksimalno število niti, ki jih OpenMP program lahko uporabi. Nikjer pa ne piše, da bo program vedno porabil vse niti, ki so mu na voljo.

Pri prenosu kode iz Xeon na Xeon Phi smo imeli težavo s stavkom `reduction`, saj se program na Xeon Phi ni uspešno končal, če je vseboval ta stavek. S prepisom kode in novejšim prevajalnikom se je napaka odpravila.

Dodatek A

A.1 Koda funkcije GS0

```
111 void GS0(double inputVectors[][NUM_OF_COMPONENTS], double outputVectors[][↩
        NUM_OF_COMPONENTS]){
112
113     int i,j,k;
114     //just rewrite all to output
115     for(i=0; i<NUM_OF_VECTORS; i++){
116         for(j=0; j<NUM_OF_COMPONENTS; j++){
117             outputVectors[i][j] = inputVectors[i][j];
118         }
119     }
120     //normalize first one
121     double dot = 0;
122     for(j=0; j<NUM_OF_COMPONENTS; j++){      //ZANKA 1
123         dot += outputVectors[0][j]*outputVectors[0][j];
124     }
125     dot = 1/sqrt(dot);
126     for(j=0; j<NUM_OF_COMPONENTS; j++){      //ZANKA 2
127         outputVectors[0][j] = dot*outputVectors[0][j];
128     }
129
130     //do the calculation for all the rest
131     for(i=1; i<NUM_OF_VECTORS; i++){         //ZANKA 3
132         for(j=0; j<i; j++){          //ZANKA 3.1
133             //DOT
134             dot = 0;
135             for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.1
136                 dot += inputVectors[i][k] * outputVectors[j][k];
137             }
138             //C1 - DOT(C1,O0)O0 and so on
```

```
139             for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.2
140                 outputVectors[i][k] -= dot*outputVectors[j][k];
141             }
142         }
143         //normalize
144         dot = 0;
145         for(j=0; j<NUM_OF_COMPONENTS; j++){ //ZANKA 3.2
146             dot += outputVectors[i][j]*outputVectors[i][j];
147         }
148         dot = 1/sqrt(dot);
149         for(j=0; j<NUM_OF_COMPONENTS; j++){ //ZANKA 3.3
150             outputVectors[i][j] = dot*outputVectors[i][j];
151         }
152     }
153 }
```

A.2 Koda funkcije GS1

```
257 void GS1(double inputVectors[][NUM_OF_COMPONENTS], double outputVectors[][↩
        NUM_OF_COMPONENTS]){
258 
259     int i,j,k;
260     //just rewrite all to output
261     #pragma omp parallel for \
262     shared(outputVectors) private(j)
263     for(i=0; i<NUM_OF_VECTORS; i++){
264         for(j=0; j<NUM_OF_COMPONENTS; j++){
265             outputVectors[i][j] = inputVectors[i][j];
266         }
267     }
268     //normalize first one
269     double dot = 0;
270     for(j=0; j<NUM_OF_COMPONENTS; j++){      //ZANKA 1
271         dot += outputVectors[0][j]*outputVectors[0][j];
272     }
273     dot = 1/sqrt(dot);
274     #pragma omp parallel for \
275     shared(outputVectors)
276     for(j=0; j<NUM_OF_COMPONENTS; j++){      //ZANKA 2
277         outputVectors[0][j] = dot*outputVectors[0][j];
278     }
279 
280     //do the calculation for all the rest
```

```
281     #pragma omp parallel for ordered\
282     shared(outputVectors) private(i,j,k,dot)
283     for(i=1; i<NUM_OF_VECTORS; i++){        //ZANKA 3
284         #pragma omp ordered
285         {
286             for(j=0; j<i; j++){          //ZANKA 3.1
287                 //DOT
288                 dot = 0;
289                 for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.1
290                     dot += inputVectors[i][k] * outputVectors[j][k];
291                 }
292                 //C1 - DOT(C1,O0)O0 and so on
293                 for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.2
294                     outputVectors[i][k] -= dot*outputVectors[j][k];
295                 }
296             }
297             //normalize
298             dot = 0;
299             for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 3.2
300                 dot += outputVectors[i][j]*outputVectors[i][j];
301             }
302             dot = 1/sqrt(dot);
303             for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 3.3
304                 outputVectors[i][j] = dot*outputVectors[i][j];
305             }
306         }
307     }
308 }
```

A.3 Koda funkcije GS2

```
189 void GS2(double inputVectors[][NUM_OF_COMPONENTS], double outputVectors[][↩
        NUM_OF_COMPONENTS]){
190 
191     int i,j,k, chunk = CHUNK_SIZE;
192     //just rewrite all to output
193     #pragma omp parallel for \
194     shared(outputVectors, chunk) private(j)
195     for(i=0; i<NUM_OF_VECTORS; i++){
196         for(j=0; j<NUM_OF_COMPONENTS; j++){
197             outputVectors[i][j] = inputVectors[i][j];
198         }
199     }
```

```
200     //normalize first one
201     double dot = 0;
202     #pragma omp parallel for \
203     shared(outputVectors) private(j) \
204     reduction(+:dot)
205     for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 1
206         dot += outputVectors[0][j]*outputVectors[0][j];
207     }
208     dot = 1/sqrt(dot);
209     #pragma omp parallel for \
210     shared(outputVectors, chunk)
211     for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 2
212         outputVectors[0][j] = dot*outputVectors[0][j];
213     }
214
215     //do the calculation for all the rest
216     #pragma omp parallel for ordered\
217     shared(inputVectors, outputVectors) private(j,k)
218     for(i=1; i<NUM_OF_VECTORS; i++){        //ZANKA 3
219         #pragma omp ordered
220         {
221             for(j=0; j<i; j++){         //ZANKA 3.1
222                 //DOT
223                 dot = 0;
224                 #pragma omp parallel for \
225                 shared(inputVectors, outputVectors) private(k) \
226                 reduction(+:dot)
227                 for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.1
228                     dot += inputVectors[i][k] * outputVectors[j][k];
229                 }
230
231                 #pragma omp parallel for \
232                 shared(outputVectors) private(k)
233                 //C1 - DOT(C1,O0)O0 and so on
234                 for(k=0; k<NUM_OF_COMPONENTS; k++){ //ZANKA 3.1.2
235                     outputVectors[i][k] -= dot*outputVectors[j][k];
236                 }
237             }
238             //normalize
239             dot = 0;
240             #pragma omp parallel for \
241             shared(outputVectors) private(j) \
242             reduction(+:dot)
243             for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 3.2
244                 dot += outputVectors[i][j]*outputVectors[i][j];
245             }
246
```

```
247             dot = 1/sqrt(dot);
248             #pragma omp parallel for \
249             shared(outputVectors, dot) private(j)
250             for(j=0; j<NUM_OF_COMPONENTS; j++){     //ZANKA 3.3
251                 outputVectors[i][j] = dot*outputVectors[i][j];
252             }
253         }
254     }
255 }
```

A.4 Koda funkcije GS8

```
192 void GS8(double inputVectors[][NUM_OF_VEC_AND_COMP], int numOfThreads){
193     int i,j,k;
194
195     //normalize first one
196     double dot2 = 0;
197     int firstIterations = (NUM_OF_VEC_AND_COMP/numOfThreads)*numOfThreads;
198
199     #pragma omp parallel for default(none) shared(inputVectors,↩
            firstIterations) private(i) reduction(+:dot2)
200     for(i=0; i<firstIterations; i++){           //ZANKA 1
201         dot2 += inputVectors[0][i]*inputVectors[0][i];
202     }
203
204     for(i=firstIterations; i<NUM_OF_VEC_AND_COMP; i++) {
205         dot2 += inputVectors[0][i]*inputVectors[0][i];
206     }
207
208     dot2=1/sqrt(dot2);
209     #pragma omp parallel for default(none) shared(inputVectors, ↩
            firstIterations) firstprivate(dot2) private(j)
210     for(j=0; j<firstIterations; j++){           //ZANKA 2
211         inputVectors[0][j] = dot2*inputVectors[0][j];
212     }
213
214     for(k=firstIterations; k<NUM_OF_VEC_AND_COMP; k++){
215         inputVectors[0][k] = dot2*inputVectors[0][k];
216     }
217
218     for(k=1; k<NUM_OF_VEC_AND_COMP; k++){ //out     //ZANKA 3
219         double dot1 = 0;
220
```

```
221         int firstIt = NUM_OF_VEC_AND_COMP;
222         if(NUM_OF_VEC_AND_COMP-k>=numOfThreads) {    //ce dolzina vecja ↩
                ali enaka od stevila jeder
223             firstIt = k+((NUM_OF_VEC_AND_COMP-k)%numOfThreads);
224         }
225 
226         #pragma omp parallel for default(none) shared(inputVectors, dot1, ↩
                firstIt) private(i,j) firstprivate(k)
227         for(i=NUM_OF_VEC_AND_COMP-1; i>=firstIt; i--) {          //ZANKA ↩
                3.1
228             double dot = 0;
229             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){           //ZANKA 3.1.1
230                 dot += inputVectors[i][j]*inputVectors[k-1][j];
231             }
232 
233             //odstejem
234             double dotTmp = 0;
235             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){           //ZANKA 3.1.2
236                 inputVectors[i][j] -= dot*inputVectors[k-1][j];
237                 dotTmp += inputVectors[i][j]*inputVectors[i][j];
238             }
239             //pass the right dot1 for normalization
240             if(i==k) {
241                 dot1 = dotTmp;
242             }
243         }
244 
245         for(i=firstIt-1; i>=k; i--) {
246             double dot = 0;
247             #pragma omp parallel for private(j) \
248             reduction(+:dot)
249             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){
250                 dot += inputVectors[i][j]*inputVectors[k-1][j];
251             }
252 
253             //odstejem
254             double dotTmp = 0;
255             #pragma omp parallel for shared(dot) private(j)\
256             reduction(+:dotTmp)
257             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){
258                 inputVectors[i][j] -= dot*inputVectors[k-1][j];
259                 dotTmp += inputVectors[i][j]*inputVectors[i][j];
260             }
261             //pass the right dot1 for normalization
262             if(i==k) {
263                 dot1 = dotTmp;
264             }
```

```
265         }
266
267         //normalize
268         dot1 = 1/sqrt(dot1);
269         #pragma omp parallel for default(none) shared(inputVectors,↩
                firstIterations) firstprivate(k,dot1) lastprivate(j)
270         for(j=0; j<firstIterations; j++){                //ZANKA 3.2
271             inputVectors[k][j] = dot1*inputVectors[k][j];
272         }
273         for(i=firstIterations; i<NUM_OF_VEC_AND_COMP; i++){
274             inputVectors[k][i] = dot1*inputVectors[k][i];
275         }
276     }
277 }
```

A.5 Koda funkcije GS9

```
192 void GS9(double inputVectors[][NUM_OF_VEC_AND_COMP], int numOfThreads){
193     int i,j,k;
194
195     //normalize first one
196     double dot2 = 0;
197
198     #pragma omp parallel for default(none) shared(inputVectors) private(i)↩
            reduction(+:dot2)
199     for(i=0; i<NUM_OF_VEC_AND_COMP; i++){        //ZANKA 1
200         dot2 += inputVectors[0][i]*inputVectors[0][i];
201     }
202
203     dot2=1/sqrt(dot2);
204     #pragma omp parallel for default(none) shared(inputVectors) ↩
            firstprivate(dot2) private(j)
205     for(j=0; j<NUM_OF_VEC_AND_COMP; j++){        //ZANKA 2
206         inputVectors[0][j] = dot2*inputVectors[0][j];
207     }
208
209     //like GS3 but has parallel second for loop
210     for(k=1; k<NUM_OF_VEC_AND_COMP; k++){      //ZANKA 3
211         double dot1 = 0;
212
213         int firstIt = NUM_OF_VEC_AND_COMP;
214         if(NUM_OF_VEC_AND_COMP-k>=numOfThreads) {    //ce dolzina vecja ↩
                ali enaka od stevila jeder
```

```
215             firstIt = k+((NUM_OF_VEC_AND_COMP-k)%numOfThreads);
216         }
217 
218         #pragma omp parallel for default(none) shared(inputVectors, dot1, ↩
                firstIt) private(i,j) firstprivate(k)
219         for(i=NUM_OF_VEC_AND_COMP-1; i>=firstIt; i--) {      //ZANKA 3.1
220             double dot = 0;
221             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){        //ZANKA 3.1.1
222                 dot += inputVectors[i][j]*inputVectors[k-1][j];
223             }
224 
225             //odstejem
226             double dotTmp = 0;
227             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){        //ZANKA 3.1.2
228                 inputVectors[i][j] -= dot*inputVectors[k-1][j];
229                 dotTmp += inputVectors[i][j]*inputVectors[i][j];
230             }
231             //pass the right dot1 for normalization
232             if(i==k) {
233                 dot1 = dotTmp;
234             }
235         }
236 
237         for(i=firstIt-1; i>=k; i--) {        //ZANKA 3.2
238             double dot = 0;
239             #pragma omp parallel for private(j) \
240             reduction(+:dot)
241             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){        //ZANKA 3.2.1
242                 dot += inputVectors[i][j]*inputVectors[k-1][j];
243             }
244 
245             //odstejem
246             double dotTmp = 0;
247             #pragma omp parallel for shared(dot) private(j)\
248             reduction(+:dotTmp)
249             for(j=0; j<NUM_OF_VEC_AND_COMP; j++){        //ZANKA 3.2.2
250                 inputVectors[i][j] -= dot*inputVectors[k-1][j];
251                 dotTmp += inputVectors[i][j]*inputVectors[i][j];
252             }
253             //pass the right dot1 for normalization
254             if(i==k) {
255                 dot1 = dotTmp;
256             }
257         }
258 
259         //normalize
260         dot1 = 1/sqrt(dot1);
```

```
261         #pragma omp parallel for default(none) shared(inputVectors) ↩
                firstprivate(k,dot1) lastprivate(j)
262         for(j=0; j<NUM_OF_VEC_AND_COMP; j++){       //ZANKA 3.3
263             inputVectors[k][j] = dot1*inputVectors[k][j];
264         }
265     }
266 }
```

Literatura

[1] 3num.pdf. https://www-old.math.gatech.edu/academic/courses/core/math2601/Web-notes/3num.pdf. Accessed: 2016-8-14.

[2] fgg_05.pdf. http://www-lp.fmf.uni-lj.si/plestenjak/vaje/nafgg/predavanja/fgg_05.pdf. Accessed: 2016-9-22.

[3] fgg_06.pdf. http://www-lp.fmf.uni-lj.si/plestenjak/vaje/nafgg/predavanja/fgg_06.pdf. Accessed: 2016-9-22.

[4] Intel-Xeon-Phi-Coprocessor_ProductBrief.pdf. http://download.intel.com/newsroom/kits/xeon/phi/pdfs/Intel-Xeon-Phi-Coprocessor_ProductBrief.pdf. Accessed: 2016-9-21.

[5] lecture3_slides.pdf. https://www0.maths.ox.ac.uk/system/files/coursematerial/2015/3135/71/lecture3_slides.pdf. Accessed: 2016-9-20.

[6] lecture4_slides.pdf. https://www0.maths.ox.ac.uk/system/files/coursematerial/2015/3135/110/lecture4_slides.pdf. Accessed: 2016-8-14.

[7] xeon_E5-2600.pdf. http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_E5-2600.pdf. Accessed: 2016-9-21.

[8] Raman Arora, Andrew Cotter, Karen Livescu, and Nathan Srebro. Stochastic optimization for pca and pls. In *Allerton Conference*, pages 861–868, 2012.

[9] G. Birkhoff and S.M. Lane. *A Survey of Modern Algebra*. AKP classics. Taylor & Francis, 1977.

[10] T. Brandes, A. Arnold, T. Soddemann, and D. Reith. Cpu vs. gpu - performance comparison for the gram-schmidt algorithm. *The European Physical Journal Special Topics*, 210(1):73 – 88, 2012.

[11] Amit Deshpande and Santosh Vempala. Adaptive sampling and fast low-rank matrix approximation. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 292–303. Springer, 2006.

[12] Dominique d'Humières. Multiple–relaxation–time lattice boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[13] Ömer Egecioglu and Ashok Srinivasan. Givens and householder reductions for linear least squares on a cluster of workstations. 1995.

[14] Seigo Enomoto, Yusuke Ikeda, Shiro Ise, and Satoshi Nakamura. Optimization of loudspeaker and microphone configurations for sound reproduction system based on boundary surface control principle. In *Proceedings of the 20th International Congress on Acoustics, ICA*, pages 1–7, 2010.

[15] Walter Gander. Algorithms for the qr decomposition. In *Seminar für Angewandte Mathematik: research report*, 1980.

[16] Alan George and Joseph WH Liu. Householder reflections versus givens rotations in sparse orthogonal decomposition. *Linear Algebra and its Applications*, 88:223–238, 1987.

[17] Seyed Roholah Ghodsi, Bahman Mehri, and Mohammad Taeibi-Rahni. A parallel implementation of gram-schmidt algorithm for dense linear system of equations. *International Journal of Computer Applications*, 1(7):16 – 20, 2010.

[18] Seyed Roholah Ghodsi and Mohammad Taeibi-Rahni. A novel parallel algorithm based on the gram-schmidt method for tridiagonal linear systems of equations. *Mathematical Problems in Engineering*, 1(7):1 – 17, 2010.

[19] Luc Giraud and Julien Langou. Robust selective gram-schmidt reorthogonalization. Technical report, Citeseer, 2002.

[20] Y Gong, TJ Lim, and B Farhang-Boroujeny. Adaptive least mean square cdma detection with gram-schmidt pre-processing. *IEE Proceedings-Communications*, 148(4):249–254, 2001.

[21] Jason R Green, Julius Jellinek, and R Stephen Berry. Space-time properties of gram-schmidt vectors in classical hamiltonian evolution. *Physical Review E*, 80(6):066205, 2009.

[22] M Heiskanen, T Torsti, Martti J Puska, and Risto M Nieminen. Multigrid method for electronic structure calculations. *Physical Review B*, 63(24):245106, 2001.

[23] Wm G Hoover and Carol G Hoover. Local gram–schmidt and covariant lyapunov vectors and exponents for three harmonic oscillator problems. *Communications in Nonlinear Science and Numerical Simulation*, 17(2):1043–1054, 2012.

[24] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[25] David Jurman, Marko Jankovec, Roman Kamnik, and Marko Topič. Calibration and data fusion solution for the miniature attitude and heading

reference system. *Sensors and Actuators A: Physical*, 138(2):411–420, 2007.

[26] Ute Kandler and Christian Schröder. Backward error analysis of an inexact arnoldi method using a certain gram schmidt variant. *preprint*, 2013.

[27] Takahiro Katagiri. Performance evaluation of parallel gram-schmidt reorthogonalization methods. In *International Conference on High Performance Computing for Computational Science*, pages 302–314. Springer, 2002.

[28] Steven J. Leon, Åke Björck, and Walter Gander. Gram-schmidt orthogonalization: 100 years and more. *Numerical Linear Algebra with Applications*, 1(1):492 – 532, 2013.

[29] Chen-Liang Lin, JF Wang, Chen-Yuan Chen, Cheng-Wu Chen, and CW Yen. Improving the generalization performance of rbf neural networks using a linear regression technique. *Expert Systems with Applications*, 36(10):12049–12053, 2009.

[30] Frederik Jan Lingen. Efficient gram–schmidt orthonormalisation on parallel computers. *Communications in Numerical Methods in Engineering*, 16(1):57 – 66, 2000.

[31] Qiaohua Liu. Modified gram–schmidt-based methods for block downdating the cholesky factorization. *Journal of computational and applied mathematics*, 235(8):1897–1905, 2011.

[32] Jia Lu, Guolai Yang, Hyounkyun Oh, and Albert CJ Luo. Computing lyapunov exponents of continuous dynamical systems: method of lyapunov vectors. *Chaos, Solitons & Fractals*, 23(5):1879–1892, 2005.

[33] Yoichi Matsuo and Takashi Nodera. An efficient implementation of the block gram–schmidt method. *ANZIAM Journal*, 54:476–491, 2013.

[34] T Maurer. How to pan-sharpen images using the gram-schmidt pan-sharpen method-a recipe. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1:W1, 2013.

[35] Benjamin Milde and Michael Schneider. Parallel implementation of classical gram-schmidt orthogonalization on cuda graphics cards.

[36] Suely Oliveira, Leonardo Borges, Michael Holzrichter, and Takako Soma. Analysis of different partitioning schemes for parallel gram-schmidt algorithms. *Parallel Algorithms And Application*, 14(4):293–320, 2000.

[37] Taisuke Ozaki. O (n) krylov-subspace method for large-scale ab initio electronic structure calculations. *Physical Review B*, 74(24):245101, 2006.

[38] Constantinos B Papadias and Alexandr M Kuzminskiy. Blind source separation with randomized gram-schmidt orthogonalization for short burst systems. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 5, pages V–809. IEEE, 2004.

[39] Yang Pu, Laura A Sordillo, Yuanlong Yang, and RR Alfano. Key native fluorophores analysis of human breast cancer tissues using gram–schmidt subspace method. *Optics letters*, 39(24):6787–6790, 2014.

[40] Gerard LG Sleijpen and Henk A Van der Vorst. The jacobi-davidson method for eigenvalue problems and its relation with accelerated inexact newton scheme. In *Proceedings of the Second IMACS International Symposium on Iterative Methods in Linear Algebra*. IMACS, 2006.

[41] Rodney Daryl Slone, Robert Lee, and Jin-Fa Lee. Well-conditioned asymptotic waveform evaluation for finite elements. *IEEE Transactions on Antennas and Propagation*, 51(9):2442–2447, 2003.

[42] Ondrej Slučiak, Hana Strakova, Amrkus Rupp, , and Wilfried N. Gansterer. Distributed gram-schmidt orthogonalization based on dynamic

consensus. *In Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers*, 31(3):1207 – 1211, 2012.

[43] Tetsuyuki Takahama and Setsuko Sakai. Solving nonlinear optimization problems by differential evolution with a rotation-invariant crossover operation using gram-schmidt process. In *Nature and Biologically Inspired Computing (NaBIC), 2010 Second World Congress on*, pages 526–533. IEEE, 2010.

[44] Andreś Tomaś and Vicente Hernández. Efficient gram-schmidt orthogonalization with cuda for iterative eigensolvers, 2010.

[45] Heinrich Voss. An arnoldi method for nonlinear eigenvalue problems. *BIT numerical mathematics*, 44(2):387–401, 2004.

[46] Charles M Werneth, Mallika Dhar, Khin Maung Maung, Christopher Sirola, and John W Norbury. Numerical gram–schmidt orthonormalization. *European Journal of Physics*, page 693 – 700, 2010.

[47] Takuya Yokozawa, Daisuke Takahashi, Taisuke Boku, and Mitsuhisa Sato. Parallel implementation of a recursive blocked algorithm for classical gram-schmidt orthogonalization. In *Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008), NTNU, Trondheim, Norway*, 2008.